MASTER EN SCIENCES INFORMATIQUES

Réalisation d'un compilateur CCS dans les langages MEC et Toupie

Kreusch, Patrick-Pierre

Award date:
1994

Awarding institution:
Universite de Namur

Année académique 1993-1994

Réalisation d'un compilateur CCS dans les langages MEC et Toupie

par
Patrick-Pierre Kreusch

Mémoire
présenté en vue de
l'obtention du diplôme de
Licencié et Maître en informatique

Promoteur : Baudouin Le Charlier
Maître de stage : Antoine Rauzy

Je tiens à remercier ceux qui, de près ou de loin, m'ont aidé à mener à bien cette fastidieuse tâche que fut la réalisation de ce mémoire. Merci à mon promoteur, Baudouin Le Charlier, pour son suivi minutieux lors de la rédaction de ce mémoire. Merci à Antoine Rauzy pour tout ce qu'il m'a appris ainsi que pour sa disponibilité face à mes nombreuses questions. Merci à Denis Zampunieris pour ses conseils et éclaircissements. Enfin, je tiens à remercier tout spécialement ma mère ainsi que Christophe pour leur « modeste » mais ô combien indispensable collaboration.

A « l'homme de la rue »
qui a guidé ma plume tant bien que mal
tout au long de la rédaction de ce mémoire.

Résumé

Dans ce mémoire, nous traitons de la réalisation d'un compilateur CCS dans les langages MEC et Toupie. Nous présentons, comment à partir d'un texte rédigé en CCS, nous extrayons sa sémantique pour la traduire dans les syntaxes de MEC et Toupie.

CCS, Calculus on Communicating Systems, est une algèbre de processus qui permet la description et l'analyse du comportement de processus et de systèmes concurrents. Il utilise la notion d'actions complémentaires pour synchroniser les processus.

Toupie est un langage de programmation sous contraintes sur les domaines symboliques finis basé sur le μ-calcul, mais c'est aussi un model-checker sur les domaines finis. A partir d'un problème donné, Toupie construit un modèle dans lequel les contraintes caractérisent des relations entre les variables et donne ensuite la possibilité de vérifier des propriétés du modèle.

MEC, tout comme CCS, sert à la modélisation et à l'analyse de systèmes concurrents. Il utilise des vecteurs de synchronisation pour régler les interactions entre les processus d'un système.

Abstract

In this thesis, we present the implementation of a compiler of CCS into MEC and Toupie. From a text written in CCS, we show how we capture its semantic to translate it into Toupie and MEC syntaxes.

CCS is a Calculus on Communicating Systems that permits to describe and analyze process behaviors and concurrent systems. It uses the notion of complementary actions to synchronize the processes of a system.

Toupie is a constraint language based on the μ-calculus over symbolic finite domains. Given a system, Toupie builds a model in which the constraints are used to characterize existing relationships between variables. Then Toupie allows to compute some properties of the system. So, it is also considered as a model-checker over finite domains.

MEC is a tool for constructing and analyzing transitions systems modelizing processes and systems of communicating processes. It uses the notion of synchronization vector to express communication between the processes of a system.

Introduction

Dans ce mémoire, nous traitons de la réalisation d'un compilateur CCS dans les langages MEC et Toupie. Celui-ci a été réalisé lors d'un stage effectué au LaBRI (Laboratoire Bordelais de Recherche en Informatique) à l'Université de Bordeaux I.

CCS, Calculus on Communicating Systems, est une algèbre de processus développées dans les années 80 par R. Milner. Elle permet la description et l'analyse du comportement de processus et de systèmes concurrents. En CCS on modélisera le comportement d'un programme par un ensemble d'états et de transitions d'états à états étiquetées par une action. De plus CCS est doté d'une syntaxe très naturelle.

Toupie est un langage de programmation sous contraintes sur les domaines symboliques finis basé sur le μ-calcul. Il est développé par A. Rauzy en collaboration avec des membres du LaBRI. Cependant les problèmes qu'on peut manipuler avec Toupie sont différents de ceux manipulés dans les langages classiques de programmation sous contraintes. A partir d'un problème donné, Toupie construit un modèle dans lequel les contraintes caractérisent des relations entre les variables. Toupie permet ensuite de vérifier si le problème possède certaines propriétés. En ce sens, Toupie est un model-checker sur les domaines finis.

MEC est aussi développé à Bordeaux par A. Arnold et sert, tout comme CCS, à la modélisation et à l'analyse de systèmes concurrents. Au contraire de CCS, MEC n'est pas seulement un formalisme théorique, un compilateur a été

développé. Son point fort vient de l'utilisation des vecteurs de synchronisation pour régler les interactions entre processus d'un système. Malheureusement, il est doté d'une syntaxe très lourde, qui rend fastidieuse l'écriture de programmes.

L'objet du mémoire était donc, à partir d'une description d'un système concurrent en CCS, d'extraire sa sémantique et de la traduire dans les syntaxes de MEC et Toupie. L'intérêt d'un tel compilateur est évident. CCS possède une syntaxe simple, naturelle et concise, ce qui n'est pas le cas de Toupie et MEC. En revanche, ceux-ci ne sont pas seulement des modèles théoriques, ils possèdent aussi une implémentation de leur langage. Donc, il est pratique de décrire un système concurrent en CCS, de le traduire en Toupie ou en MEC afin de pouvoir automatiser la vérification de propriétés du système.

Dans notre mémoire, nous avons consacré les trois premiers chapitres à la présentation des langages impliqués, à savoir CCS, Toupie et MEC. Pour chacun d'entre eux nous présentons leurs concepts de base, leur syntaxe et leur sémantique. Le quatrième chapitre est consacré à la présentation des règles de traduction, c'est-à-dire aux différentes structures utilisées pour extraire la sémantique du code CCS et la traduire en MEC et en Toupie. Dans le chapitre cinq, nous présentons les principaux algorithmes qui réalisent ce compilateur et nous prouvons leur correction. Enfin dans le dernier chapitre, nous donnons deux exemples où nous montrons toutes les phases de transformation, du code source aux codes destination.

Chapitre Premier

CCS, une algèbre de processus*

Nous présentons tout d'abord brièvement la théorie, nous montrons son utilité pour modéliser les systèmes concurrents. Nous introduisons l'ensemble des éléments de la théorie à l'aide d'un exemple. Ensuite nous en présentons plus formellement les concepts de base et nous donnons sa sémantique opérationnelle. Enfin nous parlons des modifications envisagées dans le cadre du mémoire par rapport à la théorie originale.

1.1 Présentation générale

CCS ou Calculus in Communicating Systems est une algèbre de processus qui a été proposée par R. Milner il y a quelques années. Celle-ci offre un modèle mathématique pour représenter des systèmes communicants (on dira aussi systèmes concurrents) ainsi que pour analyser le comportement de tels systèmes.

Pour analyser le comportement d'un programme séquentiel, il semble naturel de le considérer comme une fonction mathématique de changement d'état de la mémoire. En effet, étant donnée la fonction qui correspond à ce programme si on connaît l'état de départ, on en déduit facilement l'état final. Mais une telle manière de procéder suppose que le programme est le seul à s'exécuter sur la machine et donc qu'il dispose de la mémoire pour lui seul.

Mais supposons maintenant que nous ayons plusieurs programmes qui s'exécutent en même temps sur la machine, ceux-ci partagent la même mémoire

---

* Rédigé d'après [Bou93] et [Mil89]

et peuvent donc aller changer les valeurs d'autres programmes, ils peuvent interférer. Dans ce cas la théorie 'fonctionnelle' n'est plus valable. On remarque donc que deux programmes ayant la même sémantique en l'absence d'interférence, peuvent se comporter de manière tout à fait différente lorsqu'ils interfèrent. C'est le problème de la concurrence.

Communication et concurrence sont des notions complémentaires et essentielles dans la compréhension de systèmes dynamiques complexes. D'une part, un tel système est composé de plusieurs parties agissant chacune en concurrences ou indépendamment les unes des autres ; d'autre part un système complexe possède une certaine unité réalisée par la communication entre ses différentes parties.

Derrière ces notions se cache la supposition implicite que tous les éléments d'un système ont leur propre identité qui persiste au cours du temps, nous appellerons ces éléments des agents. Nous utiliserons ce terme de manière très générale. Ainsi il pourra s'appliquer aussi bien aux systèmes composés qu'aux systèmes atomiques. Il désignera tout à la fois le système ou bien un de ses composants. Plus formellement, le terme agent désignera tout système dont le comportement est composé d'actions discrètes. On utilisera aussi le terme processus pour désigner un tel système. Toute action d'un agent sera, soit une interaction avec ses agents voisins, on parlera alors de communication ; soit elle se déroulera indépendamment des actions des autres agents, on parlera alors de concurrence.

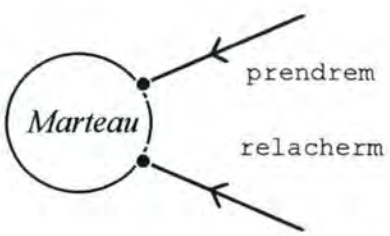
Dans cette optique, on considère que le comportement d'un système est exactement ce qui est observable. On parlera d'équivalence d'observations entre agents pour exprimer le fait que deux agents sont équivalents si et seulement si les communications qu'ils accomplissent suivent le même 'modèle' mais dont le comportement interne peut différer fortement.

C'est l'idée qui sous-tend CCS : mettre la communication au centre de la théorie. Un programme n'est plus représenté comme une fonction sur les états de la mémoire mais bien par son comportement observable (externe), par son interaction avec d'autres programmes. Ce comportement sera modélisé par un ensemble d'états et de transitions d'état à état, ces dernières étant réalisées par une action.

1.2 Modéliser la communication

Nous présentons ici les concepts et les éléments de base de la théorie à l'aide d'un exemple. Essayons de modéliser une ligne de production. Supposons que deux personnes se partagent deux outils pour fabriquer des objets. Ceux-ci sont fabriqués en assemblant une cheville dans un bloc et ce à l'aide d'un maillet ou d'un marteau. Les objets à assembler arrivent séquentiellement.

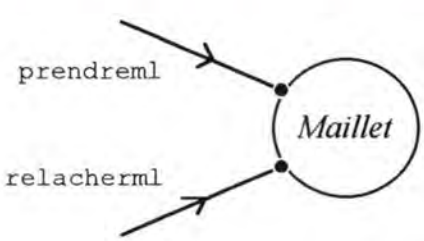
Nous considérerons comme agents de notre système, les travailleurs et les outils. Décrivons tout d'abord l'agent *Marteau*. On peut considérer celui-ci comme une ressource qui ne peut qu'être acquise ou libérée. Nous l'illustrons comme suit :



L'agent *Marteau* peut donc accomplir deux actions : `prendrem` et `relacherm`. Nous dirons qu'il est muni de deux ports que nous qualifierons ici de ports d'entrée. L'entrée est représentée sur le schéma par l'orientation des flèches (ici vers l'agent). On parle de port d'entrée pour caractériser la réception dans une communication. C'est pourquoi, par la suite, nous utiliserons indifféremment port d'entrée, action de réception et réception. Nous écrirons en CCS :

$$Marteau = \texttt{prendrem.relacherm}.Marteau$$

Donc le comportement de l'agent *Marteau* est une succession infinie d'actions `prendrem` et `relacherm`. De la même manière, nous avons pour le *Maillet* :



$$Maillet = \texttt{prendreml.relacherml}.Maillet$$

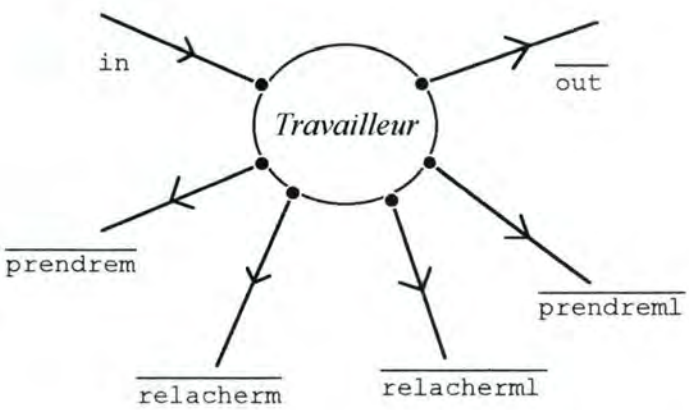
Il est évident que le comportement de l'agent *Marteau* est exactement le même que celui de l'agent *Maillet*, à condition de remplacer `prendrem`, `relacherm` par `prendreml`, `relacherml`. Nous verrons par après comment représenter ceux-ci par un seul agent.

Le comportement de l'agent *Travailleur* sera décrit comme suit : celui-ci reçoit les éléments à assembler, il utilise ensuite un des deux outils pour les assembler puis il termine et remet l'objet construit dans la chaîne.

On peut schématiser les différentes étapes dans l'activité du travailleur par les états suivants :

Debut	il a reçu les objets
Utilisemarteau	il emploie le marteau pour son travail
Utilisemaillet	il emploie le maillet pour son travail
Utiliseoutil	il emploie un des outils pour son travail
Terminer	il a terminé son travail

Nous le schématisons ainsi :



Les actions `in` et $\overline{\texttt{out}}$ symbolisant le début et la fin du travail (c'est-à-dire, le moment où le travailleur reçoit les éléments à assembler et le moment où il remet l'objet assemblé dans la chaîne). Les flèches orientées vers l'extérieur représentent l'émission dans une communication. Dans ce cas, on parlera de port de sortie.

En CCS, on écrira :

Travailleur = `in`.*Debut*
Debut = *Utiliseoutil*

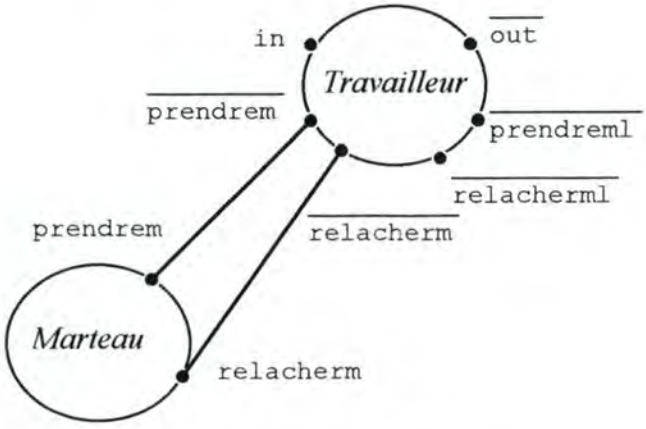
$$Utiliseoutil = Utilisemarteau + Utilisemaillet$$
$$Utilisemarteau = \overline{\texttt{prendrem}}.\overline{\texttt{relacherm}}.Terminer$$
$$Utilisemaillet = \overline{\texttt{prendreml}}.\overline{\texttt{relacherml}}.Terminer$$
$$Terminer = \overline{\texttt{out}}.Travailleur$$

Jusqu'à présent, nous avons déjà introduit deux opérateurs : le '.', que l'on appelle opérateur de préfixage et qui symbolise les actions accomplies en séquence, et le '+' ou opérateur de sommation, représentant l'alternative entre différentes actions.

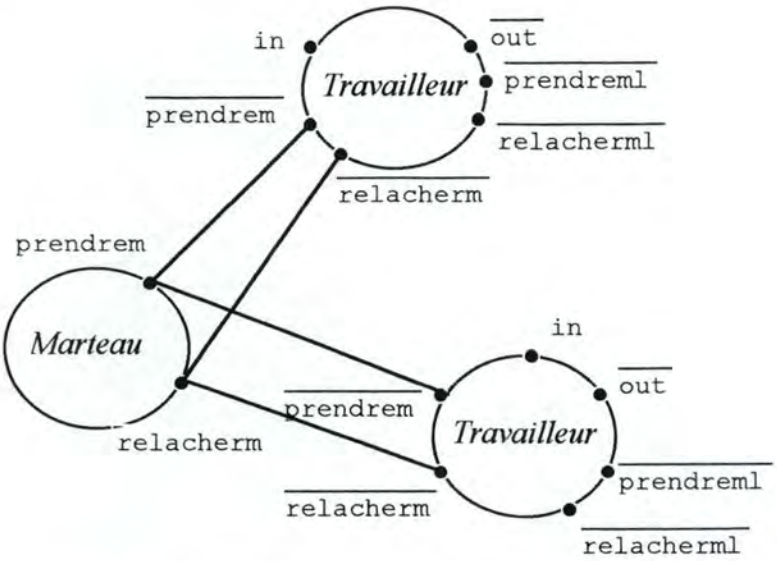
Pour compléter notre modélisation, nous devons encore introduire trois opérateurs. Commençons par l'opérateur parallèle (ou encore composition). Nous écrirons

Travailleur | | *Marteau*

pour désigner le système dans lequel *Travailleur* et *Marteau* peuvent agir indépendamment l'un de l'autre, mais dans lequel ils peuvent aussi interagir au travers de leurs ports (actions) complémentaires (`prendrem` et $\overline{\texttt{prendrem}}$ par exemple). Notre système peut être illustré comme suit :



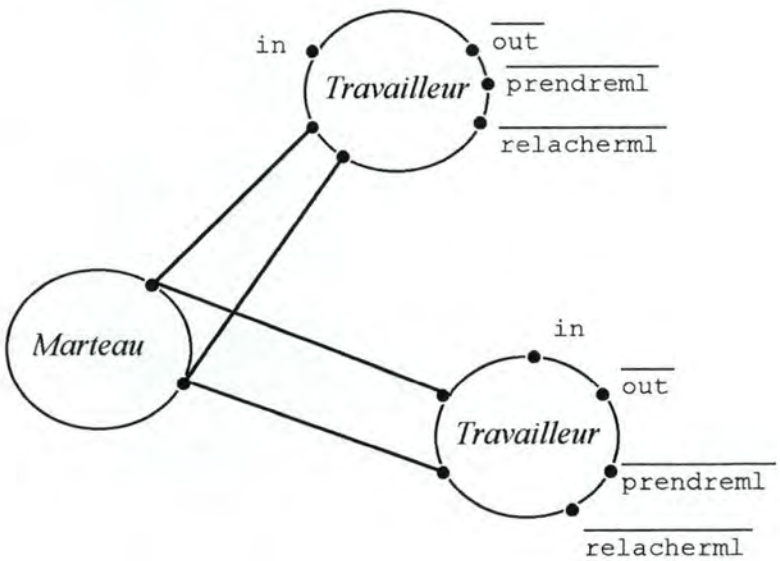
Seules ont été reliées les paires de ports (actions) complémentaires, car c'est ainsi que nous allons modéliser la communication entre deux agents : en leur faisant accomplir des actions complémentaires simultanément. En fait l'ensemble des actions de *Travailleur* | | *Marteau* est l'union de l'ensemble des actions de *Marteau* et de l'ensemble des actions de *Travailleur*. Nous dirons qu'aucune action n'a été internalisée, c'est à dire qu'on peut toujours connecter un agent quelconque à *Marteau* ou à *Travailleur*. Ainsi, si on veut illustrer que deux travailleurs se partagent le marteau, on aura :



Et on peut ajouter ainsi de suite autant d'agents que l'on veut. Au contraire, on peut vouloir internaliser un port de telle sorte qu'on ne puisse plus y connecter d'agents, pour qu'il devienne indisponible pour les communications externes. C'est ici qu'intervient notre quatrième opérateur, l'opérateur restriction. Pour internaliser les ports `prendrem` et `relacherm` de telle sorte que seulement deux travailleurs puissent se partager le marteau, par exemple, on écrira en CCS :

(*Travailleur* | | *Travailleur* | | *Marteau*)\{`prendrem`,`relacherm`}

On peut le schématiser de la manière suivante :



En n'indiquant plus les noms des ports qui ont été reliés, on signifie qu'ils sont devenus indisponibles pour les agents extérieurs au système *Travailleur* || *Travailleur* || *Marteau*. Mais cela va plus loin, une telle notation interdit à tout agent du système d'effectuer seul une des actions `prendrem` et `relacherm` ainsi que leur action complémentaire. L'agent pourra l'effectuer si un autre agent tente d'effectuer l'action complémentaire en même temps. Nous établissons ainsi sur quelles actions doivent se faire les communications au sein d'un système.

Nous pouvons maintenant décrire notre atelier tel qu'il a été défini au début. Cela donne :

Atelier = (*Travailleur* ||
Travailleur ||
Marteau ||
Maillet
)
\{prendrem,relacherm,prendreml,relacherml}

Nous avons dit plus haut que les agents *Marteau* et *Maillet* se comportent de la même manière. Nous allons maintenant voir comment les représenter en tant qu'instances du même agent. A cet effet nous introduisons la notion de renommage.

Si l est une étiquette quelconque, alors $\bar{l}$ est son complément. Nous dirons qu'une fonction f allant d'un ensemble d'étiquettes dans un autre ensemble d'étiquettes est une fonction de renommage si elle respecte le complément, c-à-d si $\forall l: f(l) = l' \Rightarrow f(\bar{l}) = \bar{l}'$. L'opérateur de renommage a pour effet de renommer les ports de l'agent auquel il s'applique.

Il semble évident que, de par leur comportement, les agents *Marteau* et *Maillet* peuvent être considérés comme des sémaphores (au sens où Dijkstra les a définis). Le comportement d'un sémaphore peut être écrit de la sorte :

Sem = p . v.*Sem*

Alors, grâce à l'opérateur de renommage, on va pouvoir écrire :

Marteau = *Sem*[prendrem/p,relacherm/v]
Maillet = *Sem*[prendreml/p,relacherml/v]

On peut alors réécrire la définition de notre atelier :

```
Atelier = ( Travailleur ||
            Travailleur ||
            Sem[prendrem/p,relacherm/v] ||
            Sem[prendreml/p,relacherml/v]
          )
          \{prendrem,relacherm,prendreml,relacherml}
```

L'algèbre de processus définie par Milner est donc, à la base, une syntaxe composée de constantes de processus, de variables de processus et d'un ensemble d'opérateurs algébriques qui permettent de construire, à partir de processus simples, des processus plus complexes.

1.3 Notions de base

Nous définissons ici les éléments de la théorie de manière plus formelle.

1.3.1 Agent

Un agent est un système dont le comportement est composé d'actions discrètes. Ces actions font passer l'agent d'un état dans un autre. On appelle transition le passage d'un état à un autre sous l'effet d'une action. Le comportement d'un agent est donc décrit par un ensemble d'états, un ensemble de transitions et un ensemble d'actions qui réalisent ces transitions.

1.3.2 Synchronisation

Nous venons d'introduire l'algèbre de Milner de manière assez informelle. Nous avons décrit la communication comme une action indivisible entre deux agents. Mais des communications comme celle pour signaler que l'on prend le marteau par exemple, n'implique pas un échange de données. C'est pourquoi nous préférons les appeler synchronisations. Nous utiliserons le terme communication lorsqu'il y aura échange de données entre les agents.

1.3.3 Action et transition

Soit $\mathcal{A}$ = {a,b,c,...} un ensemble infini d'actions élémentaires et $\overline{\mathcal{A}}$ l'ensemble des actions conjuguées de $\mathcal{A}$, tel que si $a \in \mathcal{A}$ alors $\overline{a} \in \overline{\mathcal{A}}$. On introduit une notion d'actions conjuguées afin de modéliser la communication le long d'un canal (une forme symbolisant l'émission, l'autre la réception). On note $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$

l'ensemble des étiquettes. L'ensemble $\mathcal{L}$ comprend presque toutes les actions qu'un agent peut effectuer. En effet nous allons voir que $\mathcal{L}$ comprend encore une action supplémentaire.

Un agent est caractérisé par un ensemble d'états. Une transition d'état à état est accomplie par une action, on écrira donc une transition de la manière suivante :

$$P \xrightarrow{1} Q$$

qui dénote le fait qu'après l'exécution de l'action `1`, l'agent P se comporte comme l'agent Q, donc que $P = 1.Q$.

Pour donner un sens à l'opérateur parallèle, nous devons déterminer quelles transitions sont possibles pour un agent composé $P \mid\mid Q$ en termes de transitions possibles pour P et Q séparément. Prenons un exemple. Soient deux agents A et B :

$$A = \mathtt{a}.A' \qquad B = \mathtt{c}.B'$$
$$A' = \overline{\mathtt{c}}.A \qquad B' = \overline{\mathtt{b}}.B$$

Si A peut faire une action seul, alors il peut aussi la faire dans le contexte $A \mid\mid B$ sans interférer avec B. On a donc :

$$A \xrightarrow{\mathtt{a}} A' \Rightarrow A \parallel B \xrightarrow{\mathtt{a}} A' \parallel B$$

Si on veut maintenant représenter la communication qui est déduite des actions $A' \xrightarrow{\overline{\mathtt{c}}} A$ et $B \xrightarrow{\mathtt{c}} B'$, il faut se rappeler qu'une communication consiste en des actions simultanées par les deux parties. On a donc :

$$A' \xrightarrow{\overline{\mathtt{c}}} A \text{ et } B \xrightarrow{\mathtt{c}} B' \Rightarrow A' \parallel B \xrightarrow{?} A \parallel B'$$

qui exprime l'idée que A et B changent simultanément. Nous supposons que '?' représente une action interne effectuée par l'agent $A' \mid\mid B$. De plus, la même action interne peut être effectuée par toute paire d'actions complémentaires $(\mathtt{b},\overline{\mathtt{b}})$ des éléments de l'agent composé. Nous allons donc introduire dans la théorie une action supplémentaire, une action interne, que nous noterons τ, pour modéliser de telles synchronisations. On pourra écrire

$$A' \xrightarrow{\overline{\mathtt{c}}} A \text{ et } B \xrightarrow{\mathtt{c}} B' \Rightarrow A' \parallel B \xrightarrow{\tau} A \parallel B'$$

Nous pouvons enfin définir l'ensemble des actions *Act* qu'un agent peut effectuer,

$$Act = \mathcal{L} \cup \{\tau\}$$

Si on note f une fonction de renommage, on étendra f à *Act* en décrétant que $f(\tau) = \tau$.

1.3.4 Opérateurs

A la base, CCS est composé d'actions et d'opérateurs algébriques. Ces derniers permettent de mettre en forme le comportement d'un agent à partir des actions. Partant d'un agent simple, les opérateurs nous donnent la possibilité de modéliser des agents plus complexes ainsi que des systèmes concurrents. CCS possède cinq opérateurs :

1. L'**opérateur de préfixage**, noté '.', il symbolise les actions accomplies en séquence. Ainsi on écrit Q = a.P pour représenter le fait que l'agent Q exécute l'action a puis se comporte comme l'agent P.

2. L'**opérateur de sommation**, noté '+', il symbolise l'alternative entre des actions. L'agent Q = a.P + b.c représente l'agent qui, soit exécute l'action a puis se comporte comme l'agent P, soit exécute les actions b et c en séquence.

3. L'**opérateur parallèle**, noté '| |', il symbolise l'agent résultant de l'exécution en parallèle de plusieurs agents. Ces différents agents peuvent interagir entre eux ou s'exécuter indépendamment l'un de l'autre. On écrit Q = A || B pour représenter le fait que Q est un agent dont le comportement résulte de l'exécution simultanée des agents A et B.

4. L'**opérateur restriction**, noté '\' ou '\{...}', il symbolise l'agent auquel on interdit d'accomplir certaines actions. On écrit P\a (ou P\{a, . . . , z}) pour signifier que l'agent P peut effectuer toute action b distincte de a ; il n'est pas autorisé à effectuer l'action a, ni son complémentaire $\overline{a}$ de manière visible. Par contre, il est possible qu'il effectue l'action τ et que celle-ci soit issue d'une communication engageant les actions a et $\overline{a}$.

5. L'**opérateur renommage**, noté '[...]', il symbolise l'agent dont on a renommmé certaines actions. Une fonction de renommage est une fonction $f : \mathcal{L} \rightarrow \mathcal{L}$ telle que $f(\overline{a}) = \overline{f(a)}$. On écrit P[a_1/a] pour représenter l'agent P dont l'action a a été renommé en a_1.

1.4 Sémantique

Comme tout programme définit un processus et donc un système de transitions par l'enchaînement de ses exécutions, nous allons donner la sémantique opérationnelle des opérateurs de CCS. Il s'agit en fait de règle de réécriture des termes. Cette réécriture est conditionnée par les actions exécutables à partir d'un terme. Rappelons qu'on note $P \xrightarrow{a} P'$ pour signifier le fait que l'agent P exécute l'action a puis se comporte comme l'agent P' ($P' = a.Q$). L'algèbre CCS contient l'ensemble des processus que l'on peut obtenir au moyen de la syntaxe ci-après, où P, Q sont des processus, a et b des actions de *Act*.

préfixage : $a.P$ dénote le processus qui peut faire l'action a et ensuite se comporter comme le processus P. Le système de transition associé est donné par la règle :

$$\frac{}{a.P \xrightarrow{a} P}$$

somme : $P + Q$ est le processus qui se comporte exclusivement soit comme le processus P, soit comme Q. Deux règles correspondent à cet opérateur, chacune exprimant une des deux possibilités :

$$\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'}$$

parallèle : $P \mid\mid Q$ est un processus qui peut se comporter de trois manières différentes, comme l'indiquent les règles suivantes :

$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \qquad \frac{Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\bar{a}} P \parallel Q'}$$

$$\frac{P \xrightarrow{a} P',\ Q \xrightarrow{\bar{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

restriction : $P\backslash$ (ou $P\backslash\{...\}$) est un processus qui ne peut effectuer certaines actions de manière visible. La règle pour cet opérateur est :

$$\frac{P \xrightarrow{b} P' \,,\; b \notin \{a, \overline{a}\}}{P \setminus a \xrightarrow{b} P' \setminus a}$$

renommage : $P[f]$, où f est une fonction de renommage, a pour règle :

$$\frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{f(a)} P'[f]}$$

1.5 Restrictions apportées à CCS

Dans le cadre de notre mémoire nous avons apporté quelques petites modifications par rapport à l'algèbre initiale telle qu'elle a été définie par Milner. Ces changement portent autant sur la syntaxe que sur la sémantique.

1.5.1 Au niveau de la sémantique

1) Nous allons tout d'abord introduire le processus NIL. C'est le plus élémentaire des processus. Il caractérise le processus vide qui ne fait aucune action. Le système de transitions associé est un système sans état et sans transition. Aucune règle ne lui est associée.

2) Milner autorisait d'écrire des choses telles que

$$Process = ((R + (a.P)) \mid\mid (b.Q))\backslash L$$

où P, Q, R sont des processus (agents)
où L est un ensemble d'actions $\{a,...,z,...\}$
où a et b sont des actions

Nous allons faire une distinction entre un processus et une synchronisation . Un processus décrit le comportement d'un agent atomique. Il utilise uniquement les opérateurs sommation et préfixage ainsi que les symboles de parenthèse. Si nous reprenons l'atelier que nous avons modélisé au début, les agents *Marteau*, *Maillet* et *Utiliseoutil* en sont de bons exemples.

Une synchronisation, quant à elle, décrit le comportement d'un agent composé. Elle utilise uniquement les opérateurs parallèle, renommage et restriction. L'opérateur parallèle est un opérateur binaire faisant uniquement intervenir des processus. En ce qui concerne l'opérateur restriction, l'ensemble des actions devra être décrit explicitement. L'agent *Atelier* peut être considéré

comme une synchronisation. On écrira donc l'exemple ci-dessus de la manière suivante :

P = ... S = a.P

Q = ... $T = R + S$

R = ... U = b.Q

Process = (T || U)\{a,...,z,...}

3) Pour les processus, nous introduisons une priorité des opérateurs : l'opérateur préfixage est plus prioritaire que l'opérateur somme. Pour les synchronisations, on impose que l'opérande de l'opérateur renommage soit parenthèsé.

1.5.2 Au niveau syntaxique

1) Nous avons décidé de distinguer syntaxiquement les processus des synchronisations. Pour cela, on écrira :

Maillet <= `prendreml.relacherml`.*Maillet* pour les processus

Atelier = (*Travailleur* || *Travailleur* ||*Marteau* || *Maillet*)
\{prendrem,relacherm,prendreml,relacherml}
pour les synchronisations

2) Au niveau des actions nous allons préfixer les actions symbolisant l'émission par '!' et celles symbolisant la réception par '?'. Ainsi `prendrem` devient `?prendrem` et $\overline{\texttt{prendrem}}$ devient `!prendrem` .

3) Nous imposons que les noms de processus et de synchronisations commencent par une lettre majuscule, ceux des actions par une lettre minuscule.

1.6 La syntaxe BNF

Nous donnons ici la syntaxe BNF de CCS tel qu'elle a été envisagée dans le cadre du mémoire.

```
<CCS_program> ::= {<CCS_declaration>}

<CCS_declaration>
        ::= <process_name> = <synchronized_product_description>
        ::= <process_name> <= <process_description>
```

```
<synchronized_product_description>
            ::= (<synchronized_process_list>)\
                                    <synchronization_action_list>

<synchronized_process_list>
        ::= <synchronized_product_description>
                                    ||<synchronized_process_list>
        ::= <relabeled_process>||<synchronized_process_list>
        ::= <synchronized_product_description>
        ::= <relabeled_process>

<relabeled_process> ::= <process_name>
                    ::= <process_name>[<relabeling_list>]

<process_description> ::= <alternatives>

<alternatives> ::= <action_list> + <alternatives>
               ::= <action_list>

<action_list> ::= <action>.<action_list>
              ::= <process_name>.<action_list>
              ::= (<alternatives>).<action_list>
              ::= <action>
              ::= <process_name>
              ::= (<alternatives>)

<action> ::= tau
         ::= <observable_action>

<observable_action> ::= !<action_name>
                    ::= ?<action_name>

<relabeling_list> ::= <substitution>
                  ::= <substitution>,<relabeling_list>

<substitution> ::= <observable_action>/<observable_action>

<synchronization_action_list> ::= <action_name>
                              ::= {<action_name_list>}

<action_name_list> ::= <action_name>,<action_name_list>

<process_name>::= string beginning with an upper case letter
              ::= NIL

<action_name>::= string beginning with a lower case letter
```

Chapitre 2

TOUPIE, un model-checker sur les domaines finis*

Dans ce chapitre, nous présentons Toupie. Nous donnons tout d'abord une présentation générale en le définissant comme un model-checker sur les domaines finis et en montrant sur quels éléments il se base : le μ-calcul et les diagrammes de décision binaires. Ensuite nous passons en revue sa syntaxe ainsi que sa sémantique. Nous présentons alors les diagrammes de décision binaire tels qu'ils ont été présentés originellement puis nous les étendons aux domaines finis. Enfin nous donnons la syntaxe BNF de Toupie.

1. Introduction

Toupie, qui est actuellement en développement à Bordeaux, est un langage de programmation sous contraintes (CLP) sur des domaines symboliques finis ($\mathcal{FD}$). Il est basé sur le μ-calcul et sur une extension des diagrammes de décision. Pour cette raison, on le considère comme faisant partie de la famille des langages logiques sous contraintes sur les domaines finis, CLP($\mathcal{FD}$). Toupie en possède la plupart des éléments de base (ensemble de prédicats prédéfinis, domaine d'interprétation, classe de contraintes). Néanmoins, Toupie ne possède pas la calculabilité des CLPs.

En plus des diverses fonctionnalités offertes par les langages classiques de programmation sous contraintes, en Toupie on peut définir des

---

* Rédigé d'après [CCRM93] et [CR93a] et [CR93b]

relations comme étant des points fixes d'équations. Il est aussi possible de quantifier les formules.

Le μ-calcul est un langage qui permet la description de machines à états finis. Cela consiste à introduire un opérateur du plus petit point fixe (μ) et à modéliser le comportement de ces machines au moyen de formules booléennes.Du point de vue technique, les relations sont encodées sous formes de diagrammes de décision. Ce formalisme, proposé par R. Bryant, permet de coder les relations de manière très compacte. De plus toute opération sur ceux-ci est aisément et rapidement réalisable.

Les problèmes qui peuvent être manipulés en Toupie sont de nature différente de ceux manipulés dans les langages classiques CLP($\mathcal{FD}$). Pour la plupart, ces langages sont conçus dans le but de trouver une solution à un problème possédant des contraintes et parfois afin de trouver le meilleur étant donné une fonction objectif.

En Toupie, on définit d'abord un problème dans lequel les contraintes sont utilisées pour caractériser les relations existant entre les variables. Toupie permet ensuite de vérifier certaines propriétés du problème. En effet, partant d'un programme, Toupie en construit le modèle (au sens logique du terme). Comme le domaine d'interprétation est fini, le système formel ainsi construit est décidable. Ce qui est capital, car cela signifie que l'on peut déduire certaines propriétés de ce système en un nombre fini d'étapes. En ce sens, on qualifiera Toupie de model-checker sur des domaines finis.

Par rapport aux langages classiques, Toupie gagne en expressivité et en efficacité. Toupie est particulièrement apte à modéliser et résoudre des problèmes tels que l'implémentation d'algorithme d'interprétation abstraite pour des langages logiques ainsi que le calcul de propriétés des automates finis.

2. Syntaxe

Nous nous attachons ici à donner la syntaxe des programmes que l'on peut écrire avec Toupie.

En Toupie, il y a trois catégories syntaxiques : les *termes*, les *formules* et les *définitions de prédicats*. Un programme Toupie est un ensemble de définitions d'équations possédant différents symboles de prédicats comme tête ou des arités différentes. Une requête Toupie quant à elle, est une formule.

2.1 Les termes

Les objets de base que l'on manipule dans Toupie sont appelés des termes. Un terme peut être une *constante*, un *domaine* ou encore une *variable*.

2.1.1 Les constantes :

Les symboles de constante sont soit des *entiers positifs* (2, 15, 453, ...), soit des *constantes symboliques*, c'est à dire tout identificateur dans [A..Z][a..zA..Z0..9_]* (telles que a, z, toto,a_constante, ...).

2.1.2 Les domaines :

Un domaine est un ensemble de constantes dans lequel seront interprétées les variables. La déclaration d'un domaine peut prendre les formes suivantes :

- `domain {k1,k2, ..., kr}` où les `ki` sont des constantes.
- `domain i..j` où `i` et `j` sont des entiers positifs tels que `i<j`.

Il est possible de donner un nom à un domaine de la manière suivante :

- `let toto {a,b,c,d,e}`

2.1.3 Les variables :

Les symboles de variable sont les identificateurs dans [A..Z][a..zA..Z0..9_]* (A_variable,Toto, ...).

Toute variable d'un programme Toupie doit avoir un domaine d'interprétation. Ce domaine doit être déclaré avec la première occurrence de la variable. Cette déclaration est de la forme `X : domain` où `domain` est un domaine quelconque où un nom de domaine déjà déclaré. La portée d'une variable est limitée à la définition du point fixe ou à la requête.

2.2 Les formules

Les formules peuvent être de la forme suivante :

- Les deux constantes booléennes 0 et 1.
- `(X in domain)` où `X` est une variable et `domain` représente un domaine quelconque.

- `(X1=X2)` ou `(X1=k)` ou `(X1#X2)` ou `(X1#k)`, `X1` et `X2` étant des variables, `k` un symbole de constante appartenant au domaine d'interprétation.
- `P(X1,...,Xn)` où `P` est une variable de prédicat n-aire et `X1, ..., Xn` sont des variables individuelles.
- `~f, (f & g), (f | g), (f <=> g), (f <#> g), (f => g)` où `f` et `g` sont des formules et `~`, `&`, `|`, `<=>`, `<#>`, `=>` représentent les connecteurs logiques $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\otimes$, $\Rightarrow$.
- `@(i,[f1,...,fn])` où `i` est un entier positif et `f1, ..., fn` sont des formules. Une telle notation exprimant le fait que ce connecteur sera satisfait si et seulement si au moins `i fi` sont satisfaits.
- `#(i,j,[f1,...,fn])` où `i` est un entier positif et `f1, ..., fn` sont fes formules. Ce connecteur sera satisfait si et seulement si au moins `i` et au plus `j fi` sont satisfaits.
- forall `X1,...,Xn f` ou `exist X1,...,Xn f` où `X1, ..., Xn` sont des variables et f est une formule.

2.3 Les définitions de prédicats

Les définitions de prédicats s'écrivent comme suit :

- `P(X1, ..., Xn) += f`
- `P(X1, ..., Xn) -= f`

où `P` est une variable de prédicat n-aire et `X1`, ..., `Xn` sont des variables individuelles et f est une formule. Le symbole += (respectivement -=) représente la définition d'un plus petit point fixe (respectivement, un plus grand point fixe).

Ainsi qu'on le fait en Prolog, on supposera que les variables non quantifiées apparaissant dans le membre de droite d'une définition d'un point fixe sont existentiellement quantifiées au début de ce même membre.

2.4 L'ordonnancement des variables

Pour des raisons inhérentes aux structures de données internes, Toupie exige que les variables d'une définition d'un point fixe ou d'une formule soient ordonnées. Par défaut, les variable sont ordonnées selon leur ordre d'apparition. Mais, pour des raisons d'efficacité (cfr le point 5 sur les BDDs) l'utilisateur peut vouloir imposer son ordre à lui. Un index peut donc être déclaré avec la première occurrence de la variable, avec la syntaxe suivante :

- `X@i` où `X` est la variable et `i` ,un entier positif, est son index

2.5 Exemple

Nous examinons ici la syntaxe d'un programme Toupie à partir de deux exemples bien connus : le problème du loup, de la chèvre et du chou, et le jeu de Nim.

2.5.1 Le loup, la chèvre et le chou

Un batelier (M) veut traverser la rivière avec son loup (W), sa chèvre (G) et son chou (C). Or son bateau n'a que deux places, de plus il ne peut laisser le loup avec la chèvre (W=G) et la chèvre avec le chou (G=C). Le programme suivant résout le problème.

```
domain {left,right}
     % symbolise les deux rives de la riviere

% reachable decrit tous les etats possibles, c-a-d toutes les
% positions possibles de l'homme (M), du loup (W), de la chevre
% (G) et du chou (C).

reachable(M,W,G,C) += (
     % etat initial, tout le monde est sur la rive gauche
     ((M=left) & (W=left) & (G=left) & (C=left))
   | (
        ( % les traversees possibles
          (reachable(M2,W,G,C) & (M2#M))
                    % l'homme seul
        | (reachable(M2,W2,G,C) & (M2=W2) & (M=W) & (M2#M))
                    % l'homme et le loup
        | (reachable(M2,W,G2,C) & (M2=G2) & (M=G) & (M2#M))
                    % l'homme et la chevre
        | (reachable(M2,W,G,C2) & (M2=C2) & (M=C) & (M2#M))
                    % l'homme et le chou
        )
      &
        % exclut les etats interdits
        ~(((W=G) | (G=C)) & (M#G))
     )
)
```

Exemple de requêtes :

```
reachable(right,right,right,right) ?

1
```

Cette requête permet de savoir si le problème est soluble. Le « `1` » indique que c'est le cas.

```
reachable(Man,Wolf,Goat,Cabbage) ?

{Man=left, Wolf=left, Goat=left}
{Man=left, Wolf=left, Goat=right, Cabbage=left}
{Man=left, Wolf=right, Goat=left}
{Man=right, Wolf=left, Goat=right}
{Man=right, Wolf=right, Goat=left, Cabbage=right}
{Man=right, Wolf=right, Goat=right}
```

Cette requête nous donne tous les positions possibles (autorisée par le problème) de l'homme, de la chèvre et du chou.

2.5.2 Le jeu de Nim

Le jeu commence avec N lignes numérotées de 1 à N et la ligne i contient 2*i-1 allumettes. A chaque tour, le joueur qui a la main prend autant d'allumettes qu'il le désire dans une et une seule des lignes puis il passe la main. Le gagnant est le joueur qui prend la dernière allumette.
Modélisons ce problème pour N=3.

A chaque ligne on associe un automate

```
let label {e,m}
     % e = ne rien prendre
     % m = prendre un nombre qcq d'allumettes

let state1 0..1

line1(S:state1,L:label,T:state1) += (
    ((L=e) & (S=T)
  | ((S=1) & (L=m) & (T=0)
)

let state2 0..3

line2(S:state2,L:label,T:state2) += (
    ((L=e) & (S=T)
  | ((S=1) & (L=m) & (T=0)
          % reste une allumette
  | ((S=2) & (L=m) & (T in 0..1))
          % reste deux allumettes
  | ((S=3) & (L=m) & (T in 0..2))
          % etat initial, reste trois allumettes
)

let state3 0..5

line3(S:state3,L:label,T:state3) += (
    ((L=e) & (S=T)
  | ((S=1) & (L=m) & (T=0)
```

```
  | ((S=2) & (L=m) & (T in 0..1))
  | ((S=3) & (L=m) & (T in 0..2))
  | ((S=4) & (L=m) & (T in 0..3))
  | ((S=5) & (L=m) & (T in 0..4))
)

synchronizator(L1:label,L2:label,L3:label) += (
    ((L1=m) & (L2=e) & (L3=e))
  | ((L1=e) & (L2=m) & (L3=e))
  | ((L1=e) & (L2=e) & (L3=m))
)
          % indique qu'on ne peut prendre des allumettes que
          % dans une seule ligne a la fois

edge(S1:state1,T1:state1,
     S2:state2,T2:state2,
     S3:state3,T3:state3)

    +=

  exist L1:label,
        L2:label,
        L3:label

    (
        line1(S1,L1,T1)
      & line2(S2,L2,T2)
      & line3(S3,L3,T3)
      & synchronizator(L1,L2,L3)
    )

reachable(T1:state1,T2:state2,T3:state3) += (
    ((T1=1) & (T2=3) & (T3=5))
  |
     exist S1:state1,
           S2:state2,
           S3:state3

     (
        reachable(S1,S2,S3)
      & edge(S1,T1,S2,T2,S3,T3)
     )
)
```

3. Sémantique

La sémantique d'une formule en Toupie est déterminée par rapport à une structure $S = \langle Const, \mathcal{V} \rangle$ où *Const* est le domaine d'interprétation défini au début du programme, $\mathcal{V}$ est un ensemble dénombrable de variables comprenant toutes les variables du programme.

Définition 1.1 [Assignation de variable]

Une assignation d'une variable est une fonction de l'ensemble $\mathcal{V}$ dans l'ensemble *Const*.

Définition 1.2 [Relation]

Une relation sur S est une fonction de $\mathcal{V} \rightarrow Const$ dans $\mathcal{B}$, où $X \rightarrow Y$ représente l'ensemble des fonctions de X dans Y et où $\mathcal{B}$ est l'ensemble des valeurs booléennes.

Définition 1.3 [Interprétation d'une variable de prédicat]

Une interprétation d'une variable de prédicat est une fonction de Pr dans $(\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$, où Pr est l'ensemble des prédicats figurant dans le programme et $\mathbb{N}$ est l'ensemble des naturels.

La sémantique d'une formule Toupie est donc une relation et la sémantique d'un prédicat est une fonction de $(\mathbb{N} \rightarrow Const)$ dans $\mathcal{B}$.

Un programme Toupie assigne une signification à un ensemble de symboles de prédicats Pr. La sémantique du programme est définie comme le plus petit point fixe de la transformation $\mathcal{T}$. Définissons maintenant deux ensembles :

$\mathcal{PR}$: l'ensemble Pr $\rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$ des interprétations de variables de prédicats

$\mathcal{RE}$: l'ensemble $(\mathcal{V} \rightarrow Const) \rightarrow \mathcal{B}$ des relations.

Un programme Toupie définit une transformation continue :

$$\mathcal{T} : \mathcal{PR} \rightarrow \mathcal{PR}$$

Chaque formule définit une fonction :

$$\mathcal{T}[\![\mathrm{f}]\!] : \mathcal{PR} \rightarrow \mathcal{RE}$$

Et chaque équation définit une fonction :

$$\mathcal{T}[\![\mathrm{Eq}]\!] : \mathcal{PR} \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$$

Définition 1.4

Soit la fonction $f : A \rightarrow B$

Soient $a_1, \ldots, a_n$ des éléments distincts de A

Soient b_1, ..., b_n des éléments quelconques de B

On notera

$$f[a_1/b_1, ...,a_n/b_n]$$

la fonction g : $A \rightarrow B$ telle que $g(a_i)=b_i$ $(1 \leq i \leq n)$ et $g(a)=f(a)$ $(\forall a \notin \{a_1,...,a_n\})$.

Nous sommes maintenant capable de définir la sémantique de la fonction $\mathcal{T}$. Convenons encore de noter π une interprétation de variable de prédicat, α une assignation de variable et σ un élément de $(\mathbb{N} \rightarrow Const)$. Nous définissons alors $\mathcal{T}$ inductivement sur la structure d'une formule de la manière suivante :

$\mathcal{T}[1]\ \pi\ \alpha \equiv 1$ et $\mathcal{T}[0]\ \pi\ \alpha \equiv 0$.
$\mathcal{T}[X_i = X_j]\ \pi\ \alpha \equiv \alpha(X_i) = \alpha(X_j)$.
$\mathcal{T}[X_i = k]\ \pi\ \alpha \equiv \alpha(X_i) = k$.
$\mathcal{T}[\sim g]\ \pi\ \alpha \equiv \neg\mathcal{T}[g]\ \pi\ \alpha$
$\mathcal{T}[f \mid g]\ \pi\ \alpha \equiv \mathcal{T}[f]\ \pi\ \alpha \vee \mathcal{T}[g]\ \pi\ \alpha$.
$\mathcal{T}[f \,\&\, g]\ \pi\ \alpha \equiv \mathcal{T}[f]\ \pi\ \alpha \wedge \mathcal{T}[g]\ \pi\ \alpha$.
$\mathcal{T}[\forall\ X\ f]\ \pi\ \alpha \equiv \wedge_{k \in Const}(\mathcal{T}[f]\ \pi\ \alpha[X/k])$.
$\mathcal{T}[\exists\ X\ f]\ \pi\ \alpha \equiv \vee_{k \in Const}(\mathcal{T}[f]\ \pi\ \alpha[X/k])$.
$\mathcal{T}[P(X_{i1},...,X_{ir})]\ \pi\ \alpha \equiv \pi(P)(\alpha(X_{i1}),...,\alpha(X_{ir}))$.
$\mathcal{T}[P(X_1,...,X_r)+=f]\ \pi\ \sigma \equiv \mathcal{T}[f]\ \pi[X_1/\sigma(1),...,X_n/\sigma(n)]$.

Finalement, la transformation associée au programme s'écrit

$$\mathcal{T}[Eq_1,...,Eq_n]\ \pi \equiv \pi[p_1/\mathcal{T}[Eq_1]\ \pi,...,p_n/\mathcal{T}[Eq_n]\ \pi]$$

où les p_i sont les prédicats définis dans les équations Eq_i.

Définition 1.5 [Dénotation d'une formule]
Soit P un programme Toupie.
Soit f une formule Toupie.
Soit D l'ensemble des variables libres de f.
On appellera la dénotation de f dans P, la fonction $\mathcal{D}[f] : (D \rightarrow Const) \rightarrow \mathcal{B}$, telle que

$$\forall\ \alpha\ \in (D \rightarrow Const), \qquad \mathcal{D}[f]\alpha = \mathcal{T}[f](\mu(\mathcal{T}[P]))\alpha'$$

où α' est toute assignation de variable telle que

$$\alpha' X = \alpha X \, (\forall X \in D)$$

4. Implémentation

Nous présentons maintenant les diagrammes de décision binaires, la structure de représentation interne des relations. Nous voyons tout d'abord les diagrammes tels qu'ils ont été proposés à l'origine, ensuite nous les généralisons aux domaines finis. Nous nous attachons aussi à montrer leur efficacité.

4.1 Les diagrammes de décision binaires*

Les diagrammes de décision binaire (BDD) sont un formalisme de représentation et de manipulation des fonctions booléennes qui fut introduit en 1986 par R. Bryant. C'est une structure très efficace car elle possède sous certaines conditions des propriétés de canonicité.

4.1.1 Formules propositionnelles et formules booléennes

Avec Toupie, nous travaillons dans le cadre de la logique propositionnelle quantifiée, tandis que les BDDs s'appliquent aux fonctions booléennes. Mais on peut montrer l'équivalence entre fonction (propositionnelle) et formule booléenne. On notera $\mathcal{B}$={**0**,**1**} l'ensemble des booléens où **0** tient pour « FAUX » et **1** tient pour « VRAI ». On appellera $\mathcal{V}$, un ensemble dénombrable de variables propositionnelles.

Définition 1.6

Pour toute formule f de la logique propositionnelle, on associe de manière unique une fonction booléenne notée f également, telle que $f : \mathcal{B}^n \rightarrow \mathcal{B}$, où n est le nombre de variables propositionnelles de la formule f, de telle sorte que, pour toute interprétation $I : \mathcal{V} \rightarrow \mathcal{B}$, on ait

$$f \models I$$
$$\Leftrightarrow$$
$$f(I(x_1),I(x_2),...,I(x_n)) = \mathbf{1}$$

* Ce point a été rédigé d'après [Bry86] et [Bou93]

4.1.2 Décomposition de Shannon des fonctions booléennes

Définition 1.7 [Expansion de Shannon]

Soit f une formule propositionnelle.

On appelle expansion de Shannon de f par rapport à la variable propositionnelle x_i le couple de formules

$$f_0 = f[0 \leftarrow x_i]$$
$$f_1 = f[1 \leftarrow x_i]$$

où « $0 \leftarrow x_i$ » se lit « x_i est remplacé par 0 ».

Cette expansion a la propriété suivante :

$$f = (\neg x_i \wedge f_0) \vee (x_i \wedge f_1)$$

Si on suppose que f s'exprime sur un ensemble $X = \{x_1, x_2, \ldots, x_n\}$ de variables propositionnelles, cela se traduit sur les fonctions booléennes associées par

$$f(x_1, x_2, \ldots, x_n) = (\neg x_i \wedge f_0(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)) \vee (x_i \wedge f_1(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n))$$

Théorème 1.1

Soit f : $\mathcal{B}^n \rightarrow \mathcal{B}$ une fonction booléenne sur les variables $X = \{x_1, x_2, \ldots, x_n\}$.

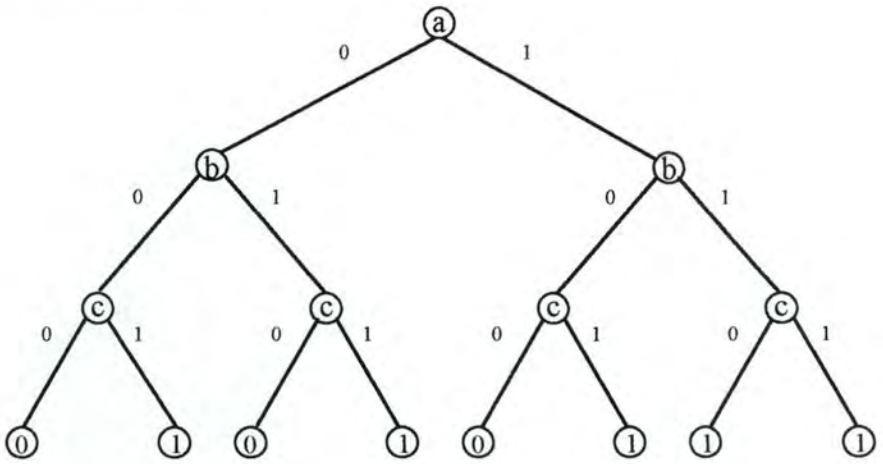
$\forall\, x_i \in X$, $\exists\, (f_0, f_1)$ de $\mathcal{B}^{n-1} \rightarrow \mathcal{B}$ sur les variables $X \setminus \{x_i\}$, tel que

$$f = (\neg x_i \wedge f_0) \vee (x_i \wedge f_1)$$

Une expansion complète d'une formule consiste en une itération de l'expansion de Shannon sur f_0 et f_1 sur une variable de $X \setminus \{x_i\}$, et ceci récursivement sur toute les variables. On obtient alors un arbre binaire dont les noeuds sont étiquetés par les variables du domaine de la fonction et les feuilles, par les deux constantes booléennes **0** et **1**. Chaque noeud possède deux arcs dont l'un est étiqueté par **0** et l'autre par **1**. Cet arbre n'est rien d'autre que la table de vérité de la formule qui se lit en suivant un chemin dans l'arbre partant de la racine, où chaque variable s'interprète selon l'étiquette de l'arc emprunté pour arriver à un de ses fils. Le résultat de l'interprétation est donné aux feuilles. On convient de prendre le fils droit (respectivement gauche) comme étant l'interprétation de la variable à **1** (respectivement **0**).

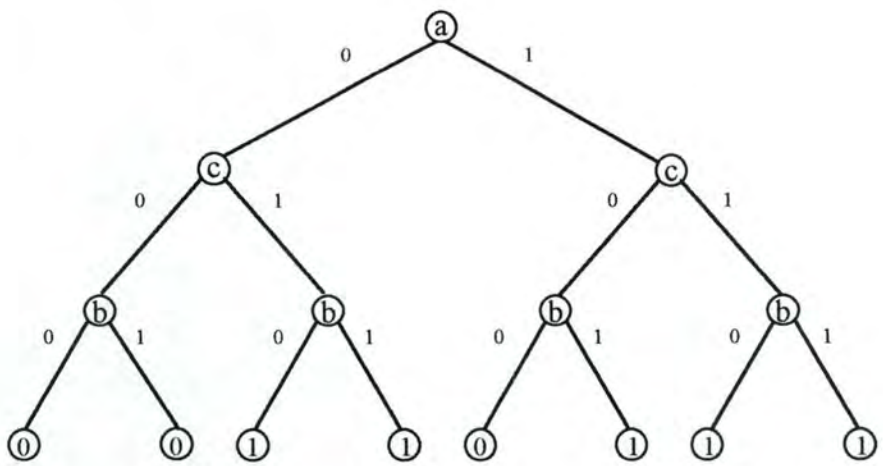
Exemple :
Soit la fonction f(a,b,c)=(a ∧ b) ∨ c

On la représente par l'arbre suivant



Cette expansion totale n'est évidemment pas unique et dépend de l'ordre dans lequel sont choisies les variables par rapport auxquelles on fait l'expansion. Ainsi dans l'exemple ci-dessus, on a pris les variables dans l'ordre a, b puis c.

Si on avait d'abord pris la variable a, puis la variable c et enfin b, on aurait eu l'arbre ci-dessous.



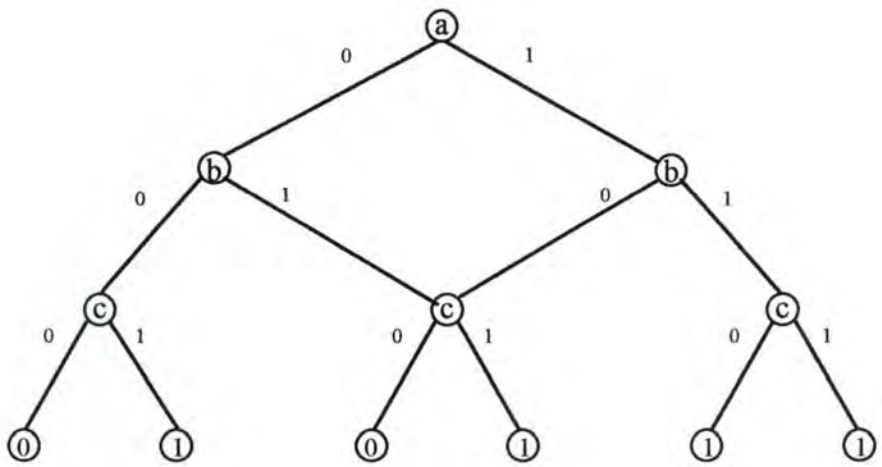
Néanmoins, pour un ordre fixé des variables du domaine, il existe une expansion complète de Shannon qui est canonique. Deux formules sémantiquement équivalentes possèdent le même arbre de Shannon.

4.1.3 De l'arbre de Shannon au diagramme de décision binaire

Les BDDs sont issus des arbres de Shannon. En fait, ils sont le résultats de deux transformations qui réduisent ces derniers tout en conservant la sémantique de la formule dans l'arbre et la propriété de canonicité.

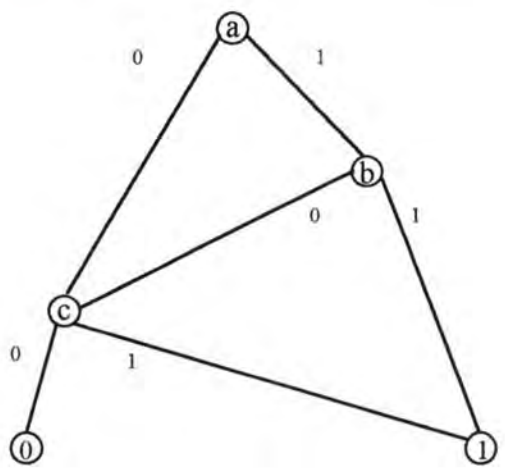
A. Partage des sous-arbres

Cette première transformation consiste à compacter l'arbre de façon à mettre en commun des sous-arbres égaux. On obtient alors un graphe acyclique orienté (DAG). En reprenant l'exemple du point précédent, on obtient



B. Elimination des noeuds redondants

Cette deuxième transformation consiste à éliminer les noeuds (variables) dont la valeur n'influe pas sur la valeur de la fonction, donc si ses deux sous-arbres fils sont égaux. Poursuivant sur notre exemple, nous obtenons finalement le BDD



Nous donnons maintenant une définition formelle d'un diagramme de décision binaire.

Définition 1.8 [Diagramme de décision binaire]

Soient V un ensemble de variables et $<$ un ordre total sur les variables de V. Un diagramme de décision binaire ordonné F est un graphe acyclique orienté, tel que :

- F a deux feuilles : **0** et **1**.
- Chaque noeud non terminal est étiqueté par une variable $X \in V$ et possède deux arcs orientés vers l'extérieur étiquetés par **0** et **1**.
- Si un noeud étiqueté par la variable Y est accessible à partir d'un noeud étiqueté par la variable X, alors $X < Y$.

4.1.4 Propriétés des BDDs

Les BDDs sont une forme canonique graphique de représentation des fonctions booléennes modulo un ordre fixé des variables que l'on utilise lors de l'expansion complète de Shannon. Comme on a pu le constater sur notre exemple, les BDDs réduisent considérablement l'arbre de Shannon initial, et ce dans bien des cas. C'est grâce à ce pouvoir de réduction que les BDDs se sont avérés comme étant une représentation des plus compactes comparativement aux autres formes normales connues. Mais leur point fort tient au niveau de l'efficacité des opérations booléennes ou autres que l'on peut opérer sur ceux-ci afin de manipuler les fonctions booléennes. En effet, la plupart des opérations sont polynomiales sur le nombre de noeuds.

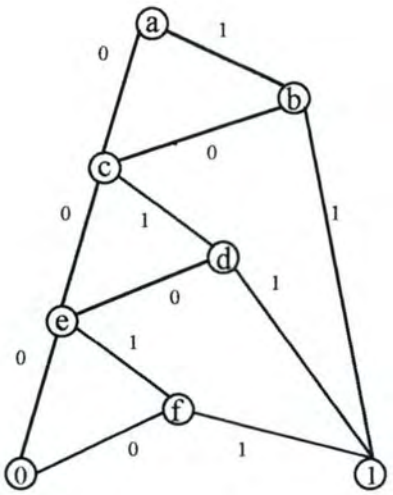
4.1.5 Ordonnancement et efficacité

Nous avons dit que les BDDs, ainsi que les arbres de Shannon dont ils sont issus, sont une représentation canonique modulo un certain ordre des variables. Pour les arbres de Shannon, nous avons vu que si le graphe changeait lorsque l'ordre des variables changeait, sa taille, quant à elle, ne changeait pas. En ce qui concerne les diagrammes de décision binaires, l'ordonnancement des variables est cruciale et détermine entièrement l'efficacité de la structure. Nous l'illustrons par un exemple.

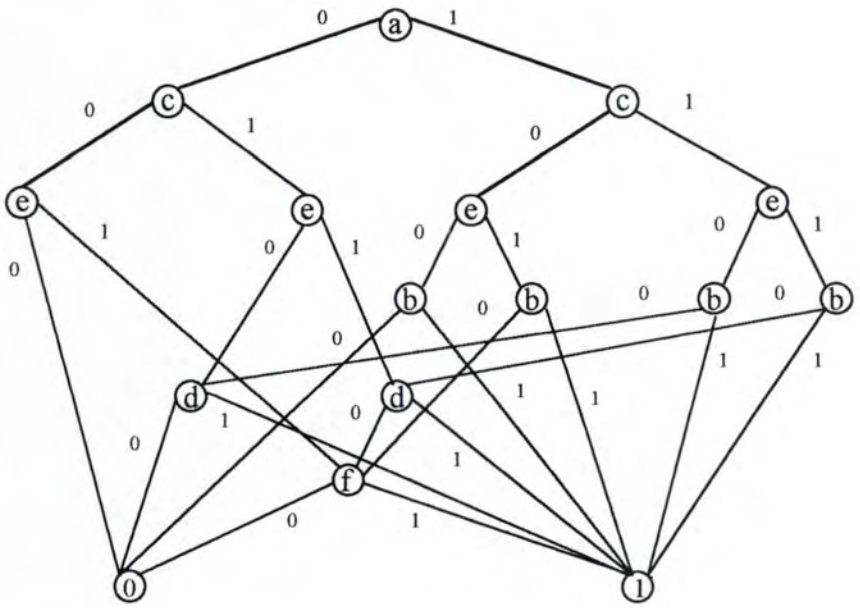
Exemple :

Soit la formule $f(a,b,c,d,e,f) = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$

1. Prenons a<b<c<d<e<f comme ordre.



2. Prenons a<c<e<b<d<f comme ordre.



L'exemple le montre clairement, alors que dans le premier choix le nombre de noeuds est de 8, dans le second, il est de 16.

Si on généralise pour des formules du même type (forme normale disjonctive) à $2n$ éléments, notre premier choix nous donnera un graphe à $2n+2$ noeuds, tandis que le deuxième nous donnera un graphe à 2^{n+1} noeuds. L'ordre des variables peut donc avoir des conséquences désastreuses sur l'efficacité des

BDDs. C'est pour pallier à cela que Toupie permet de définir son propre ordre d'évaluation des variables.

4.2 Généralisation des BDDs aux domaines finis

Les diagrammes de décision sont utilisés dans Toupie pour encoder les relations. Malheureusement ces relations ne sont pas toujours binaires, mais comme ces relations portent sur des domaines symboliques finis, il est possible de généraliser les BDDs aux domaines finis.

4.2.1 Le connecteur case

Introduisons tout d'abord le connecteur *case* qui est la généralisation de l'expansion de Shannon aux domaines finis.

Définition 1.9 [Connecteur *case*]

Soit le domaine d'interprétation $Const = \{k_1,...,k_r\}$.

Soit X une variable.

Soit $f_1,...,f_r$ des formules.

Alors

$$case(X, f_1,...,f_r) = ((X = k_1) \wedge f_1) \vee ... \vee ((X = k_r) \wedge f_r)$$

On dira qu'une formule est sous forme normale de Shannon si un des deux points suivants est vérifié :

- $f = \mathbf{0}$ ou $f = \mathbf{1}$
- $f = case(X, f_1,...,f_r)$, où X est une variable et $f_1,...,f_r$ sont des formules sous forme normale de Shannon dans lesquelles X n'apparaît pas.

Théorème 1.2

Soit $V = \{X_1,...,X_n\}$ un ensemble de variables.

Alors, pour toute relation n-aire R, il existe une formule sous forme normale de Shannon qui l'encode.

4.2.2 Les diagrammes de décision

On généralise alors la définition du BDD au domaines finis de la manière suivante :

Définition 1.10 [Diagramme de décision ordonné]

Soit le domaine d'interprétation $Const = \{k_1,...,k_r\}$.

Soient V un ensemble de variables et $<$ un ordre total sur les variables de V.

Un diagramme de décision ordonné F est un graphe acyclique orienté, tel que :

- F a deux feuilles : **0** et **1**.
- Chaque noeud non terminal est étiqueté par une variable $X \in V$ et possède au moins deux arcs et au plus r, orientés vers l'extérieur, étiquetés par $k_1,...,k_r$.
- Si un noeud étiqueté par la variable Y est accessible à partir d'un noeud étiqueté par la variable X, alors $X < Y$.
- Si deux noeuds F et G sont distincts, soit ils sont étiquetés par des variables différentes, soit leurs sous-arbres diffèrent par au moins un noeud.

La propriété de canonicité peut alors s'énoncer ainsi :

Soit R une relation n-aire sur les variables $X_1,...,X_n$.

Soit $<$ un ordre total sur ces variables.

Alors, il existe un et un seul diagramme de décision ordonné encodant R.

4.2.3 Propriétés des DAGs ordonnés

La propriété suivante montre à quel point les diagrammes de décisions ordonnés sont efficaces pour réaliser des opérations logiques sur les relations.

Théorème 1.3

Soit * un opérateur logique binaire.

Soient $p = case(X,p_1,...,p_r)$ et $q = case(X,q_1,...,q_r)$ deux formules sous forme normale de Shannon.

Soit $V = \{k_1,...,k_r\}$ le domaine d'interprétation.

Alors, on a

$$case(X,p_1,...,p_r) * case(X,q_1,...,q_r) = case(X,p_1 * q_1,...,p_r * q_r)$$

Démonstration

$$case(X,p_1,...,p_r) * case(X,q_1,...,q_r) = (X = k_1) \wedge (case(X,p_1,...,p_r) * case(X,q_1,...,q_r)) \vee (X = k_2) \wedge (case(X,p_1,...,p_r) * case(X,q_1,...,q_r))$$

$$\vee$$

$$\ldots$$

$$\vee$$

$$(X = k_r) \wedge (case(X,p_1,\ldots,p_r) * case(X,q_1,\ldots,q_r))$$

$$= (X = k_1) \wedge (p_1 * q_1) \vee \ldots \vee (X = k_r) \wedge (p_r * q_r)$$

$$= case(X,p_1 * q_1,\ldots,p_r * q_r)$$

□

De ce principe, il sera aisé de déduire une procédure effective.

4.2.4 Efficacité des diagrammes de décision

Toupie gagne en efficacité par rapport aux langages classiques pour plusieurs raisons inhérentes à l'implémentation.

Tout d'abord, il faut savoir que Toupie fonctionne en deux phases. Premièrement, il « compile » le programme que l'utilisateur lui soumet, c'est-à-dire qu'il calcule tous les points fixes. Deuxièmement, on l'interroge en lui soumettant des requêtes. Durant cette phase, plus aucun calcul n'est effectué, on se contente de comparer la requête au programme. On réalise en fait un « ET » logique entre la requête et le programme.

Ensuite, les diagrammes de décision codent les relations sur les domaines finis d'une manière très compacte au moyen du partage des sous-arbres. Ce partage est automatiquement réalisé en stockant les noeuds dans une table de haschage. Chaque fois qu'on a besoin d'un noeud $case(X,p_1,\ldots,p_r)$, on va d'abord regarder dans la table s'il s'y trouve et on le crée le cas échéant.

De plus, la propriété de canonicité permet par exemple de réduire le calcul de l'égalité de deux expressions encodées sous forme de diagramme de décision à un test d'égalité entre les deux adresses des diagrammes.

Un autre point très important qui rend les diagrammes de décision efficaces en pratique, est que la procédure de calcul utilise un mécanisme d'apprentissage : chaque fois qu'une opération $p*q$ est réalisée, le résultat est mémorisé dans une table de haschage. Donc un tel calcul n'est jamais réalisé deux fois. Cette amélioration est souvent considérable en pratique et devient de plus en plus importante au fur et à mesure que la taille du programme augmente.

5. Syntaxe BNF

Nous donnons ici la syntaxe BNF de Toupie.

```
<request> ::= <fixed-point-or-request>
          ::= <formula> ?

<fixed-point-or-request> ::= <predicate> += <formula>
                         ::= <predicate> -= <formula>
                         ::= <predicate> ?

<formula> ::= <parenthesed-formula>
          ::= <projection>
          ::= <cardinality>
          ::= <at-least>
          ::= <quantifier>
          ::= <not>
          ::= <boolean-constant>
          ::= <predicate>

<parenthesed-formula> ::= ( <atomic-relation> )
                      ::= ( <composed-formula> )

<composed-formula> ::= <n-ary-connective>
                   ::= <binary-connective>
                   ::= <formula>

<binary-connective> ::= <formula> => <formula>
                    ::= <formula> <=> <formula>
                    ::= <formula> <#> <formula>

<n-ary-connective> ::= <item-list[ <formula> | ]>
                   ::= <item-list[ <formula> & ]>

<predicate> ::= <identifier> ( item-list[ <argument> , ] )

<not> ::= ~ <formula>

<boolean-constant> ::= 0 | 1

<atomic-relation> ::= <variable> = <variable>
                  ::= <variable> # <variable>
                  ::= <variable> = <constante>
                  ::= <variable> # <constante>
                  ::= <variable> in <domain>


<quantifier> ::= exist <variable-list><formula>
             ::= forall <variable-list><formula>

<at-least> ::= @(<integer>, [ <item-list[ <formula> , ]> ])
```

```
<cardinality>
      ::= #(<integer>, <maximum>, [<item-list[<formula> , ]> ])

<projection> ::= [ <formula>; <assignement-list> ]

<assignement-list>
           ::= <variable> <- <argument> , <assignement-list>
           ::= <variable> <- <argument>

<item-list[ <item> <separator> ]
       ::= <item> <separator> <item-list[ <item> <separator> ]>
       ::= <item>

<argument> ::= <constant> | <domain> | <variable>

<domain> ::= <constant-set> | <range> | <functor>

<constant-set> ::= { <item-list[ <constant> , ]> }

<range> ::= <integer>..<integer>

<constant> ::= <integer> | <identifier>

<identifier> ::= any string of alphanum or _ letters beginning
                 with a lower case letter.

<variable> ::= any string of alphanum or _ letters beginning
               with an upper case letter.
```

Chapitre 3

MEC, un système de construction et d'analyse de systèmes de transitions*

1. Introduction

MEC est un outil servant à la construction et à l'analyse de systèmes de transitions qui modélisent des processus et des systèmes de processus concurrents.

A partir de la représentation des processus par des systèmes de transitions et à partir de la représentation des interactions entre les processus d'un système par l'ensemble de toutes les actions globales permises, MEC construit un système de transitions représentant le système concurrent comme le produit synchronisé des composants du système.

MEC permet de vérifier qu'un tel système de transitions possède certaines propriétés en calculant les ensembles d'états et les ensembles de transitions vérifiant celles-ci. Ces propriétés sont exprimées dans un langage permettant les définitions de nouveaux opérateurs logiques en tant que plus petit point fixe d'un système d'équations.

---

* Rédigé d'après [ABC94]

2. Concepts

Mec manipule trois principaux concepts : le système de transitions, l'action globale, qu'on appelle aussi vecteur de synchronisation, et le produit synchronisé.

2.1 Système de transitions

La notion de système de transitions joue un rôle important dans la description et l'étude des processus et des systèmes concurrents. Une manière simple de représenter un processus est de considérer que c'est un ensemble d'états et qu'une action change l'état courant de celui-ci. On représente alors les comportements élémentaires possibles d'un processus par des transitions. Chaque transition comprend l'état courant du processus, le nouvel état dans lequel il va entrer ainsi que le nom de l'action qui provoque ce changement.

Définition 3.1 [Système de transition/Automate]

Un système de transitions étiqueté sur un alphabet A d'actions est un t-uple $\mathcal{A} = \langle S,T,\alpha,\beta,\lambda \rangle$ où

- S est un ensemble fini d'états
- T est un ensemble fini de transitions
- $\alpha,\beta : T \rightarrow S$ sont les fonctions qui associent à chaque transition t son état d'origine $\alpha(t)$ et son état d'arrivée $\beta(t)$
- $\lambda : T \rightarrow A$ est la fonction qui étiquette chaque transition t par l'action $\lambda(t)$ qui en est la cause.

On écrira aussi l'automate $\mathcal{A}$ sur l'alphabet A sous la forme du t-uple $\langle S,T,A,\alpha,\beta,\lambda \rangle$

Nous supposons qu'il n'existe jamais deux transitions différentes avec la même étiquette et entre les deux mêmes états. Cela signifie que la fonction $\langle \alpha,\lambda,\beta \rangle : T \rightarrow S \times A \times S$ est injective. Cela veut encore dire qu'il n'est pas possible de distinguer deux transitions possédant la même cause et le même effet sur le système.

2.2 Système synchronisé

Les systèmes de transitions sont aussi utilisés pour modéliser le comportement de systèmes de processus concurrents. Les états du système sont

alors des t-uples d'états des processus qui composent celui-ci et les transitions du système sont des t-uples de transitions de ses processus.

Considérons n systèmes de transitions $\mathcal{A}_i$ sur les alphabets A_i. Supposons que ceux-ci forment un système de processus en interaction. Cela signifie qu'une certaine action dans un certain processus peut être exécutée seulement de manière simultanée avec une autre action d'un autre processus ou, au contraire, qu'elle peut être exécutée indépendamment de toute action de tout autre processus.

Nous appelons vecteur de synchronisation le vecteur $<a_1,...,a_n>$ où $a_i \in \mathcal{A}_i$. Un tel vecteur est exécuté lorsque les actions a_i sont exécutées simultanément par les n processus. Un vecteur de synchronisation est un vecteur d'actions qui décrit le fait que, lorsqu'un processus du système veut exécuter telle action, soit il doit le faire en parallèle avec telle ou telles autre(s) action(s) d'un ou d'autres processus du système ; soit il doit l'exécuter indépendamment de tout autre. Un vecteur de synchronisation décrit l'interaction ou l'absence d'interaction entre les processus d'un système concurrent. Donc les interactions entre les processus d'un système peuvent être représentées par l'ensemble de tous les vecteurs de synchronisation qui peuvent être exécutés.

Définition 3.2 [Vecteur de synchronisation]

Soit n systèmes de transitions $\mathcal{A}_i = <S_i, T_i, \alpha_i, \beta_i, \lambda_i>$ sur les alphabets A_i.

Soit $\mathcal{A} = <\mathcal{A}_1,...,\mathcal{A}_n>$ le système concurrent composé des n automates $\mathcal{A}_i$.

Alors un vecteur de synchronisation du système $\mathcal{A}$ est un vecteur de $A_1 \times ... \times A_n$, noté $<a_1,...,a_n>$ tel que :

- soit $\exists!\ i \in \{1..n\}$: a_i représente une action quelconque ($\neq e$) et $a_j = e$ $\forall\ j \in \{1..n\} \backslash i$.

  Dans ce cas le vecteur décrit une action qui ne doit être synchronisée avec aucune autre.

- soit $\exists\ i_1,...,i_l \in \{1..n\} \wedge 1 < l \leq n$: a_{i_k} représente une action quelconque ($\neq e$) et $a_j = e\ \forall\ j \in \{1..n\} \backslash \{i_1,...,i_l\}$.

  Dans ce cas le vecteur décrit une interaction entre l actions.

L'action e est l'action nulle, elle représente le fait qu'un système reste dans le même état.

Un vecteur de synchronisation est un vecteur d'actions, donc un vecteur d'étiquettes. Chaque action étiquette une transition effectuée par un processus d'un système concurrent. L'ensemble des actions d'un vecteur étiquettent en fait ce que nous appellons une transition globale, i.e. un ensemble de transitions effectuées en même temps par les processus du système.

2.2.1 Contrainte de synchronisation

Comme expliqué ci-dessus, les interactions entre les processus $\mathcal{A}_i$ d'un système sont représentés par un sous-ensemble I de $A_1 \times ... \times A_n$ qui est en fait l'ensemble des vecteurs de synchronisation et que l'on appelle aussi contrainte de synchronisation. Toute interaction qui n'est pas reprise dans la contrainte de synchronisation est donc strictement interdite dans le système.

Définition 3.3 [Contrainte de synchronisation]

Soit n systèmes de transitions $\mathcal{A}_i = <S_i,T_i,\alpha_i,\beta_i,\lambda_i>$ sur les alphabets A_i.

Soit $\mathcal{A} = <\mathcal{A}_1,...,\mathcal{A}_n>$ le système concurrent composé des n automates $\mathcal{A}_i$.

Alors la contrainte de synchronisation, notée I, associée à $\mathcal{A}$ est t.q.

$I = \{ V_i \mid V_i$ est un vecteur de synchronisation de $\mathcal{A} \}$

Or, comme $\forall i : V_i \in A_1 \times ... \times A_n \quad \Rightarrow \quad I \subseteq A_1 \times ... \times A_n$

2.2.2 Produit synchronisé

Définition 3.4 [Produit libre]

Soit un vecteur $<\mathcal{A}_1,...,\mathcal{A}_n>$ de systèmes de transitions où $\mathcal{A}_i = <S_i,T_i,\alpha_i,\beta_i,\lambda_i>$ sur l'alphabet A_i.

Le produit libre de $<\mathcal{A}_1,...,\mathcal{A}_n>$ est le système de transitions $<S,T,\alpha,\beta,\lambda>$ sur $A_1 \times ... \times A_n$ défini par

$$S = S_1 \times ... \times S_n$$
$$T = T_1 \times ... \times T_n$$
$$\alpha(t_1,...,t_n) = <\alpha_1(t_1),...,\alpha_n(t_n)>$$
$$\beta(t_1,...,t_n) = <\beta_1(t_1),...,\beta_n(t_n)>$$
$$\lambda(t_1,...,t_n) = <\lambda_1(t_1),...,\lambda_n(t_n)>$$

Dans un certain sens, le produit libre représente l'évolution du vecteur de systèmes de transitions lorsqu'aucune contrainte n'est mise sur les actions qui

peuvent être exécutées simultanément. Lorsqu'on met une contrainte de synchronisation, certaines transitions de ce produit libre n'apparaissent pas. Ce sont celles qui sont étiquetées par un vecteur d'actions qui n'est pas autorisé par la contrainte de synchronisation. Cela signifie que ces interactions ne sont pas admises dans le système. Nous pouvons alors définir le produit synchronisé.

Définition 3.5 [Produit synchronisé]

Soit un vecteur $<\mathcal{A}_1,...,\mathcal{A}_n>$ de systèmes de transitions où $\mathcal{A}_i = <S_i,T_i,\alpha_i,\beta_i,\lambda_i>$ sur l'alphabet A_i

Soit une contrainte de synchronisation I incluse dans $A_1 \times ... \times A_n$

Le produit synchronisé de $<\mathcal{A}_1,...,\mathcal{A}_n>$ en respect de I est le système de transitions $<S,T_I,\alpha,\beta,\lambda>$ sur $A_1 \times ... \times A_n$ où

- $<S,T,\alpha,\beta,\lambda>$ est le produit libre de $\mathcal{A}_1,...,\mathcal{A}_n$
- T_I est l'ensemble des transitions $t = <t_1,...,t_n>$ de T dont l'étiquette $\lambda(t) = <\lambda_1(t_1),...,\lambda_n(t_n)>$ est dans I.

2.3 Calcul de propriétés

MEC permet de vérifier qu'un système concurrent possède ou non certaines propriétés. Le mécanisme utilisé est l'exécution d'assignations. La forme générale d'une assignation est *variable := expression* comme dans les langages de programmation. La valeur prise par *expression* est soit un ensemble d'états, soit un ensemble de transitions du système.

3. Syntaxe

Nous présentons maintenant la syntaxe de MEC. Nous allons modéliser l'algorithme d'exclusion mutuelle de Peterson pour deux processus. Cet algorithme utilise trois variables booléennes globales `flag[1]`, `flag[2]` et `turn`, toutes trois initialisées à 0. L'algorithme est le suivant :

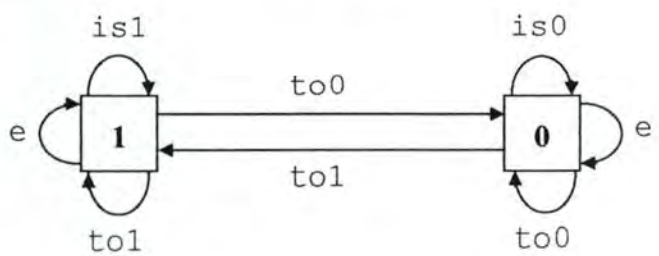
```
proc(me,other) =
while true do
begin
                0:...;
{mutexbegin}      flag[me]:=1;
                1:turn = me;
                2:WAIT(flag[other]=0 OR turn=other);
{cs}            3:...;
{mutexend}        flag[me]=0;
end
```

Dans cet algorithme `me` vaut 0 et `other` vaut 1 pour le premier processus et `me` vaut 1 et `other` 0 pour le second.

Chaque processus est modélisé par un système de transitions. Chaque variable booléenne est aussi modélisée par un système de transitions.
Une variable booléenne a deux états, notés 0 et 1, et peut effectuer les actions :

`e` qui ne fait rien
`to0` qui signifie mettre la variable à 0
`to1` qui signifie mettre la variable à 1
`is0` qui teste si la valeur de la variable est 0
`is1` qui teste si la valeur de la variable est 1

Il en résulte que le système possède huit transitions :



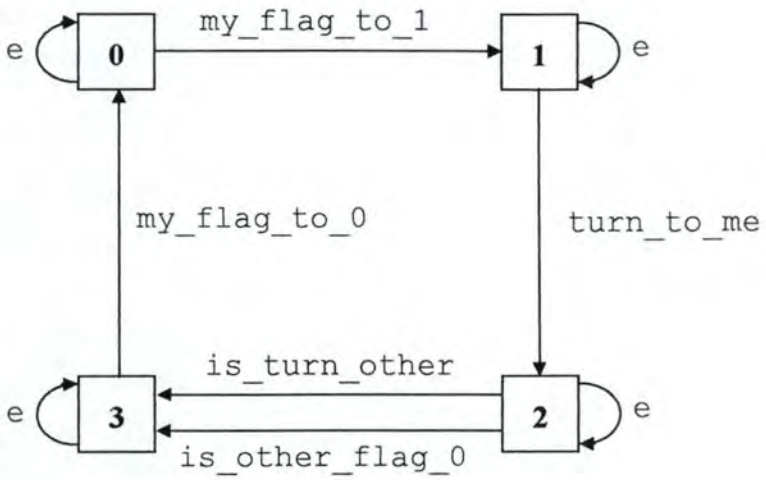
Un système de transitions sera décrit explicitement par l'ensemble de toutes ses transitions. On écrira donc en MEC :

```
transition_system b < width = 0 >;
0 |- e -> 0;
0 |- to0 -> 0;
0 |- is0 -> 0;
0 |- to1 -> 1;

1 |- e -> 1;
1 |- to1 -> 1;
1 |- is1 -> 1;
1 |- to0 -> 0;

< initial = { 0 } >.
```

Le paramètre `width = 0` indique qu'il s'agit d'un système de transitions simple, i.e. qui ne synchronise aucun processus.

Le paramètre `initial = { 0 }` indique que l'état initial du processus est l'état 0.

Le système de transitions modélisant proc est schématisé ci-dessous.



Nous écrirons donc en MEC :

```
transition_system proc < width = 0 >;
0 |- e -> 0;
0 |- my_flag_to_1 -> 1;

1 |- e -> 1;
1 |- is_other_flag_0 -> 2;
1 |- is_turn_other -> 2;

3 |- e -> 3;
3 |- my_flag_to_0 -> 0;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.
```

Les paramètres `cs = { 3 }` et `ncs = { 0 }` indiquent respectivement qu'à l'état 3 le processus est dans sa section critique et qu'à l'état 0 le processus n'est pas dans sa section critique.

Le produit synchronisé décrit le comportement du système concurrent à l'aide des vecteurs de synchronisation. Si on suppose que :

1. le système fonctionne en monoprocesseur, donc les deux processus ne pourront effectuer simultanément une action non nulle et il ne pourront être en attente en même temps.

2. Chaque action réalisée par un processus consiste à mettre à jour ou à tester une variable. Quand un processus exécute une telle action, la variable correspondante exécute l'action correspondante et les autres variables exécutent l'action nulle.

On écrira donc :

```
synchronization_system Peterson
                    < width = 5 ; list = (proc,proc,b,b,b) >;

(my_flag_to_0     . e               . to0 . e   . e  );
(my_flag_to_1     . e               . to1 . e   . e  );
(e                . my_flag_to_0    . e   . to0 . e  );
(e                . my_flag_to_1    . e   . to1 . e  );
(turn_to_me       . e               . e   . e   . to0);
(e                . turn_to_me      . e   . e   . to1);
(is_other_flag_0  . e               . e   . is0 . e  );
(e                . is_other_flag_0 . is0 . e   . e  );
(is_turn_other    . e               . e   . e   . is1);
(e                . is_turn_other   . e   . e   . is0).
```

Les paramètres `width = 5` et `list = (proc,proc,b,b,b)` indiquent le nombre de processus que le système synchronise et la liste de ces processus.

Si on considère maintenant les deux premières colonnes du système, la première (resp. deuxième) contient toutes les actions effectuées par le premier (resp. deuxième) processus, on peut voir que sur chaque ligne de ce sous-système il y a une et une seule action nulle, ce qui corrobore bien l'hypothèse un.

Si nous considérons maintenant la première ligne du système, lorsque le premier processus met son flag à 0, la première variable `b`, qui représente `flag[0]` exécute l'action `to0`. A la troisième ligne, c'est la seconde variable, qui représente `flag[1]` qui exécute `to0` lorsque le second processus exécute `my_flag_to_0`. Dans le même ordre d'idée, les actions `turn_to_me` sont exécutées simultanément avec une action exécutée par la troisième variable b qui représente `turn`. La valeur à laquelle cette variable est mise dépend du processus qui met la variable à jour. Le test `is_other_flag_0` est exécuté simultanément avec l'action `is0`, la variable booléenne exécutant cette action dépendant du processus qui réalise ce test. De même l'action `is0` ou `is1` est exécutée par la troisième variable booléenne simultanément avec le test

`is_turn_other` exécuté par le deuxième ou le premier processus. L'hypothèse deux est donc bien vérifiée.

Si nous voulons maintenant vérifier que la modélisation que nous avons faite de l'algorithme de Peterson assure bien la propriété d'exclusion mutuelle, nous devrons calculer s'il existe des états (globaux) dans lequel les deux processus sont dans leur section critique.

En MEC, on écrira `cs[i]` pour désigner l'ensemble des états dans lequel le $i^{ème}$ système de transitions du système est dans sa section critique. Pour l'exclusion mutuelle, on écrit `nok := cs[1] ∧ cs[2].`
Après l'exécution de cette commande, on peut constater que la valeur de `nok` est l'ensemble vide. Donc l'exclusion mutuelle est une propriété du système.

4. La sémantique

La sémantique de MEC est relativement évidente et apparaît clairement au travers de la syntaxe.

En effet, la sémantique d'un processus, modélisé en MEC par un système de transitions, est un système de transitions, au sens de la définition 3.1. La sémantique d'une synchronisation d'un système concurrent est, de la même manière, un produit synchronisé, au sens de la définition 3.5.

5. Comparaison MEC-CCS

CCS et MEC permettent tous deux la modélisation et l'analyse de systèmes concurrents. Néanmoins ils diffèrent fortement l'un de l'autre.

Tout d'abord, à la différence de MEC qui est non seulement un modèle théorique mais aussi pratique (on dispose d'un compilateur), CCS est un modèle purement théorique, c'est une algèbre de processus qui permet de modéliser et d'étudier le comportement de processus.

Ensuite, on utilise CCS pour la puissance de sa syntaxe. En effet la description du comportement d'un processus s'effectue assez naturellement et de manière très concise en CCS. Tandis que MEC impose de devoir expliciter beaucoup de choses lors de la description d'un système de transitions. Pour la description d'un processus, il faut explicitement décrire toutes les transitions qui composent celui-ci. Pire, concernant le produit synchronisé, il convient de décrire toutes les interactions autorisées dans le système (ce qui peut être très long pour

certains problèmes). La syntaxe de MEC est donc très lourde. En pratique il est très fastidieux de modéliser des systèmes dès que ceux-ci deviennent quelques peu complexes. Cela vient du fait que la syntaxe est calquée sur sa sémantique.

Mais, il n'empêche que la notion de vecteur de synchronisation utilisée en MEC est une notion très puissante. En CCS, on synchronise les processus sur l'exécution en parallèle d'actions complémentaires permettant de cette manière de synchroniser seulement deux processus à la fois (l'un exécutant une action, l'autre son complémentaire). Le vecteur de synchronisation par contre, permet la synchronisation de deux ou plusieurs processus à la fois, ce qui autorise la description de systèmes beaucoup plus complexes.

6. La syntaxe BNF

Nous donnons une syntaxe BNF réduite de MEC.

```
<axiome> ::= <transition_system>

<transition_system> ::=
            transition_system <identifier>
                              <attribute_zone>;
                              <transition_system_body>.

<attribute_zone> ::= < width = <integer> >

<transition_system_body> ::= <state_description_list>
                         ::= <state_description_list>
                             <set_state_zone>

<state_description_list>
               ::= <state_description> <state_description_list>
               ::= <state_description>

<state_description> ::= <state_list>;

<state_list> ::= <state>,<state_list>
             ::= <state>

<state> ::= <integer> |- <label_identifier> -> <integer>

<set_state_zone> ::= < initial = { <state> } >

<synchronization_system> ::=
          synchronization_system <identifier>
                                 <parameter_zone>;
                                 <synchronization_system_body>.

<parameter_zone> ::=
              < width = <integer> ; <transition_system_list> >;
```

```
<transition_system_list> ::= list = ( <identifier_list> )

<identifier_list> ::= <identifier>,<identifier_list>
                  ::= <identifier>

<synchronization_system_body> ::= <synchronization_vector_list>

<synchronization_vector_list>
     ::= <synchronization_vector>;<synchronization_vector_list>
     ::= <vector>

<synchronization_vector> ::= ( <label_identifier_list> )

<label_identifier_list>
                 ::= <label_identifier>.<label_identifier_list>
                 ::= <label_identifier>

<label_identifier> ::= any string of alphanum or _ letters


<state> ::= <state_identifier>

<identifier> ::= any string of alphanum or _ letters
```

Chapitre 4

Règles de traduction

Dans ce chapitre nous présentons d'une manière générale comment nous avons élaboré notre compilateur. Cela comprend d'abord le choix des structures de représentation internes, ensuite la description des étapes qui permettent d'arriver à ces structures tout en conservant la sémantique de l'objet CCS qui est traduit. Enfin nous décrivons comment traduire ces structures pour générer du code Toupie et du code MEC qui soient bien équivalents au code CCS de départ.

1. Choix d'une structure interne

Le principal problème auquel nous sommes confronté lors de la réalisation d'un compilateur est que, partant d'un texte source rédigé dans un langage quelconque, il s'agit d'exprimer dans un autre langage (et donc sous une autre forme) la « même chose » que le texte source. Plus précisément, il faut trouver la sémantique sous-jacente au texte source et l'exprimer dans le langage destination.

Or, la syntaxe CCS, tout comme la syntaxe MEC d'ailleurs, permet la description du comportement d'agents simples (processus) et d'agents composés (systèmes concurrents). Le modèle sémantique sous-jacent aux syntaxes de CCS et de MEC est le système de transitions étiquetées, qu'on appelle aussi automate. Ainsi la sémantique d'une description de processus est un automate. Mais si la sémantique d'une description de synchronisation est aussi un automate, on parlera plutôt de produit synchronisé car il règle l'exécution des processus d'un système concurrent.

En ce qui concerne Toupie, sa vocation n'est évidemment pas seulement de décrire des systèmes concurrents et sa sémantique n'est donc pas un automate. Néanmoins, ces notions d'automate et de produit synchronisé sont aisément modélisables.

Donc, le problème de la traduction peut se résumer comme suit. L'automate étant le modèle sémantique commun aux trois langages, à partir d'un texte exprimé en CCS, il faut construire les divers automates décrits par celui-ci. Pour chaque processus on construit l'automate qui lui correspond, pour chaque synchronisation, on construit le produit synchronisé qui lui correspond. Une fois la sémantique extraite du texte CCS, il faut alors la traduire, c'est-à-dire l'exprimer sous une forme différente, la forme des langages destination, à savoir Toupie et MEC.

2. Du processus à l'automate

Ayant choisi notre structure de représentation, l'automate, nous décrivons ici les différentes étapes de la transformation. Comme il s'agit bien entendu de produire un automate sémantiquement équivalent au processus décrit en CCS, nous avons dû passer par plusieurs structures intermédiaires avant d'arriver à l'automate. Ces structures sont l'arbre, un premier type d'automate puis l'automate final que nous avos appelé automate réduit.

Avant tout, il convient de rappeler qu'un automate est caractérisé par un ensemble d'états, un ensemble de transitions étiquetées et un ensemble d'étiquettes (cfr chapitre 3, définition 3.1)

2.1 Texte CCS

Point de départ de la transformation. La description d'un processus en CCS utilise les deux opérateurs binaires '.' et '+'. Les processus sont écrits en notation infixée. L'usage des parenthèses est permis. Pour ne pas alourdir les notations, nous avons introduit une priorité à chaque opérateur ('.' > '+'). Nous introduisons aussi une règle d'association à droite.

2.2 Arbre

C'est la structure qui nous a semblé la plus naturelle pour la première tranformation du processus. Elle a l'avantage d'être facilement réalisable à partir du texte CCS comme nous le montrons au chapitre suivant. D'une manière générale, un arbre est constitué d'une racine, qui contient une certaine information, et de n arbres fils.

Définition 4.1 [Arbre]

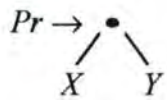
Un arbre est un t-uple A = <info,f_1,...,f_n> où

- info est le contenu de la racine
- f_1,...,f_n sont les arbres fils

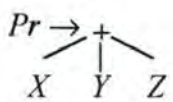
Par la suite, on écrira aussi <info,fils> où fils est un t-uple (f_1,...,f_n).

Lorsque n=0 on appelle A une feuille et on note <info,∅>.

A l'opérateur '.' on fait correspondre un arbre. Ainsi, $Pr = X.Y$ deviendra Pr = <•,X,Y> que l'on schématisera



L'opérateur '+' qui était binaire en CCS, devient maintenant un opérateur à deux ou plus de deux opérandes, on lui fait correspondre un arbre. Ainsi $Pr = X+Y+Z$ devient Pr = <+,X,Y,Z> que l'on représente par



Nous définissons maintenant de manière formelle l'équivalence sémantique entre la description CCS d'un processus et la structure d'arbre et ce par induction sur la structure de l'expression.

Définition 4.2 [Equivalence sémantique processus-arbre]

Soit Pr, une description de processus CCS.

Soit Pr*, un arbre.

Alors l'arbre Pr* est sémantiquement équivalent à la description Pr ssi

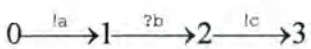
1. Si $Pr \rightarrow$ `action` et Pr* = <`action`,∅>
2. Si $Pr \rightarrow$ `process_name` et Pr* = <`process_name`,∅>
3. Si $Pr \rightarrow op_1.op_2$ et Pr* = <.,op_1*,op_2*>
4. Si $Pr \rightarrow op_1+...+op_n$ *et* Pr* = <+,op_1*,...,op_n*>

La transformation en arbre est assez naturelle avons-nous dit. Elle est nécessaire car elle nous rapproche un peu de notre but : l'automate. En effet, comme nous le montrons par la suite, nous sommes maintenant en mesure, simplement à partir de la lecture de l'arbre dans un ordre infixé, d'extraire l'automate qui y est décrit.

2.3 Automate

Il s'agit en fait d'une première structure d'automate. Celle-ci est déduite directement de l'arbre et comporte encore des éléments redondants (que nous avons introduit pour des facilités d'implémentation) par rapport à la structure à laquelle nous voulons arriver et que nous appellerons automate réduit.

Soit le processus *Pr* <= `!a.?b.!c`, nous schématiserons l'automate équivalent par

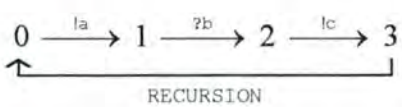


qui exprime l'idée que, partant d'un état initial 0, l'accomplissement de l'action *!a* nous fait passer dans un état 1 et ainsi de suite jusqu'à l'état final 3.

Plus formellement, l'automate équivalent sera le t-uple *Pr* = <*S*,*T*,*L*,α,β,λ> où

S = { 0,1,2,3 }
T = { 0 —!a→ 1,1 —?b→ 2,2 —!c→ 3 }
L = { `!a,?b,!c` }

Si nous avons maintenant le processus *Pr* <= `!a.?b.!c.`*Pr* qui s'appelle récursivement, nous représentons l'automate

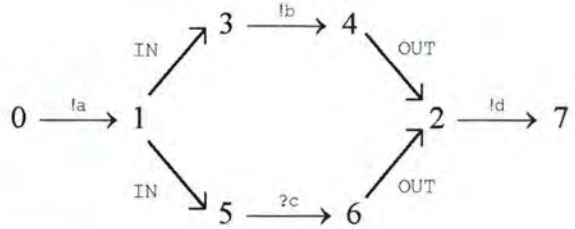


en introduisant une transition supplémentaire étiquetée par le symbole spécial `RECURSION`. Formellement, cela donne le t-uple *Pr* = <*S*,*T*,*L*,α,β,λ> où

S = { 0,1,2,3 }
T = { 0 —!a→ 1,1 —?b→ 2,2 —!c→ 3,3 —RECURSION→ 0 }

L = { !a,?b,!c,RECURSION }

Pour des raisons d'implémentation, il convient encore d'introduire deux symboles spéciaux : IN et OUT. Nous le montrons avec le processus *Pr* <= !a.(!b + ?c).!d schématisé par



et $Pr = <S,T,L,\alpha,\beta,\lambda>$ où

S = { 0,1,2,3,4,5,6,7 }

$T = \left\{ \begin{array}{l} 0 \xrightarrow{!a} 1,1 \xrightarrow{IN} 3,1 \xrightarrow{IN} 5,3 \xrightarrow{!b} 4, \\ 5 \xrightarrow{?c} 6,4 \xrightarrow{OUT} 2,6 \xrightarrow{OUT} 2,2 \xrightarrow{!d} 7 \end{array} \right\}$

L = { !a,?b,!c,IN,OUT }

L'introduction de ces transitions supplémentaires étiquetées par un symbole particulier (IN, OUT, RECURSION) est réalisée uniquement dans le but de faciliter l'implémentation. En effet, nous aurions tout aussi bien pu extraire directement l'automate tel qu'il est exprimé par l'arbre, sans toutes ces redondances.

Nous définissons l'équivalence sémantique entre un arbre et un automate par induction sur la taille de l'arbre.

Soit P un arbre contenant la description d'un processus CCS. Sans perte de généralité, nous considérons qu'un processus possède un état initial et un état final que nous notons entree(P) et sortie(P). Convenons encore de noter (x,y,z) la transition d'origine x et d'arrivée z étiquetée par y.

Définition 4.3 [Equivalence sémantique arbre-automate]

Soit P, un arbre.

1. P → feuille

A) feuille = <<action>,∅>

Alors l'automate équivalent à P est le t-uple $<S,T,L,\alpha,\beta,\lambda>$ tel que

- S = {entree(P),entree(P)+1}
- T = {(entree(P),action,entree(P)+1)}

- L = {action}
- α,β,λ sont les fonctions telles qu'elles ont été définies dans le chapitre précédent.

B) feuille=Q où Q est un processus appelant P (appel récursif)

Alors l'automate équivalent à P est le t-uple <S,T,L,α,β,λ> tel que

- S = {entree(P),entree(Q)}
- T = {(entree(P),RECURSION,entree(Q))}
- L = {RECURSION}
- α,β,λ sont les fonctions telles qu'elles ont été définies dans le chapitre précédent.

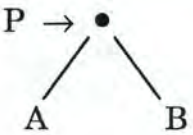
C) feuille=Q où Q n'appelle pas P.

Alors l'automate équivalent à P est le t-uple <S,T,L,α,β,λ> tel que

- S = {S_Q} où entree(P)=entree(Q)
- T = {T_Q}
- L = {L_Q}

  où S_Q, T_Q, L_Q sont les états, les transitions et les étiquettes de l'automate associé à Q.
- α,β,λ sont les fonctions telles qu'elles ont été définies dans le chapitre précédent.

2. P →



où <S_A,T_A,L_A,α_A,β_A,λ_A> et <S_B,T_B,L_B,α_B,β_B,λ_B> sont respectivement les automates de A et de B.

Alors l'automate équivalent à P est le t-uple <S,T,L,α,β,λ> tel que

- $S = S_A \cup_d S_B$
- $T = T_A \cup_d T_B$
- $L = L_A, \cup_d L_B$ où $\cup_d$ est l'union disjointe
- entree(P)=entree(A)
- sortie(B)=sortie(P)
- sortie(A)=entree(B)
- α,β,λ sont les fonctions telles qu'elles ont été définies dans le chapitre précédent.

3. P → +

X_1 ... X_n

où $<S_{X_i},T_{X_i},L_{X_i},\alpha_{X_i},\beta_{X_i},\lambda_{X_i}>$ est l'automate de X_i $\forall$ i=1...n.

Alors l'automate équivalent à P est le t-uple $<S,T,L,\alpha,\beta,\lambda>$ tel que

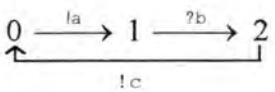
- $S = S_{X_1} \cup_d ... \cup_d S_{X_n} \cup \{\text{entree(P)},\text{sortie(P)}\}$
- $T = T_{X_1} \cup_d ... \cup_d T_{X_n} \cup \{T_I,T_O\}$ où $T_I = \bigcup_{i=1}^{n} (\text{entree(P)},\text{IN},\text{entree}(X_i))$

  $T_O = \bigcup_{i=1}^{n} (\text{sortie}(X_i),\text{OUT},\text{sortie(P)})$
- $L = L_{X_1} \cup_d ... \cup_d L_{x_n} \cup_d \{\text{IN},\text{OUT}\}$
- α,β,λ sont les fonctions telles qu'elles ont été définies dans le chapitre précédent.

2.4 Automate réduit

Nous parlons maintenant d'automate réduit car, par rapport à la structure précédente, il comporte moins ou autant de transitions, moins ou autant d'états, moins ou autant d'étiquettes. En effet, par rapport à l'automate précédent, nous avons supprimé toutes les transitions qui étaient étiquetées par un symbole spécial (IN, OUT, RECURSION) et par la même occasion nous avons fusionné certains états.

Ainsi, le processus *Pr* <= `!a.?b.!c.Pr` est maintenant schématisé par l'automate réduit



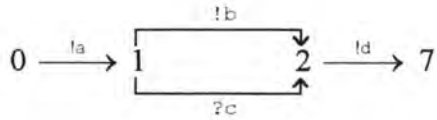
où on a supprimé la transition étiquetée par RECURSION en fusionnant son état d'arrivée sur son état de départ ce qui a fait disparaître l'état n° 3. On a maintenant l'automate réduit *Pr* = $<S,T,L,\alpha,\beta,\lambda>$ où

$S = \{ 0,1,2 \}$

$T = \{ 0 \xrightarrow{!a} 1,1 \xrightarrow{?b} 2,2 \xrightarrow{!c} 0 \}$

$L = \{$ `!a,?b,!c` $\}$

De même, le processus *Pr* <= !a.(!b + ?c).!d a comme automate réduit



et $Pr = \langle S,T,L,\alpha,\beta,\lambda \rangle$ où

$S = \{ 0,1,2,7 \}$

$T = \{ 0 \xrightarrow{!a} 1,1 \xrightarrow{!b} 2,1 \xrightarrow{?c} 2,2 \xrightarrow{!d} 7 \}$

$L = \{ !a,?b,!c \}$

Toutes les transitions étiquetées par le symbole IN ont été supprimées en fusionnant leur état source sur leur état destination et ce qui a causé la disparition des états n° 3, 5. Quant aux transitions étiquetées par OUT, leur état final a été fusionné sur leur état initial et a ainsi fait disparaître les états n° 4 et 6.

Définition 4.4 [Equivalence sémantique automate-automate réduit]

Soit un automate $P = \langle S_P,T_P,L_P,\alpha_P,\beta_P,\lambda_P \rangle$

L'automate réduit $Q = \langle S_Q,T_Q,L_Q,\alpha_Q,\beta_Q,\lambda_Q \rangle$ est sémantiquement équivalent à l'automate P

$$\Leftrightarrow$$

$S_Q = S_P \backslash R1_Q$

où $R1_Q = \{ s \in S_P \mid \exists\, t \in T_P :$
$((\alpha_P(t)=s \wedge (\lambda_P(t)=OUT \vee \lambda_P(t)=RECURSION)) \vee$
$(\beta_P(t)=s \wedge \lambda_P(t)=in)) \}$

$T_Q = T_P \backslash R2_Q$ où $R2_Q = \{ t \in T_P \mid \lambda_P(t) \in \{IN,OUT,RECURSION\} \}$

$L_Q = L_P \backslash R3_Q$ où $R3_Q = \{IN,OUT,RECURSION\}$

$\alpha_Q = \alpha_P \mid T_Q$

$\beta_Q = \beta_P \mid T_Q$

$\lambda_Q = \lambda_P \mid T_Q$

où '|' signifie 'restreint à' (i.e. $\alpha_Q : T_Q \rightarrow S_Q$)

Lors de cette transformation, on a supprimé toutes les transitions étiquetées par un symbole spécial en fusionnant leur état source avec leur état destination. Tous les états qui ont été fusionné sont supprimé de l'automate ainsi que les étiquettes spéciales.

Au point précédent nous avions déjà une structure d'automate équivalente au processus de départ mais avec de la redondance. Nous aurions pu nous arrêter là et traduire cet automate en MEC ou en Toupie puisque nous avons montré qu'il correspondait à la sémantique du processus de départ. Alors pourquoi cette réduction, ce compactage de l'automate ?

Tout simplement pour des raisons d'efficacité. En effet, une fois modélisé en Toupie (en MEC), la compilation de ces automates ainsi que les calculs effectués sur ceux-ci seront d'autant plus efficaces, c'est-à-dire rapides et économes en place mémoire, que l'automate sera petit. Il convient donc d'avoir la représentation la plus compacte possible d'un automate.

2.5 Texte Toupie et texte MEC

C'est la dernière étape, il s'agit de traduire l'automate obtenu en Toupie ou en MEC. On parle ici de traduction et non plus de transformation car on ne modifie plus la structure, contrairement aux étapes précédentes, on se contente de la « lire » d'une certaine manière et de lui donner l'aspect désiré, c'est-à-dire une syntaxe particulière.

2.5.1 Texte Toupie

Il s'agit maintenant de « traduire » l'automate, d'exprimer sa sémantique en Toupie. Cela revient à décrire le comportement de celui-ci en fonction de ses transitions, de ses étiquettes et de ses états.

A l'ensemble d'états ainsi qu'à l'ensemble d'étiquettes on fait correspondre un domaine Toupie. Tandis que pour l'ensemble des transitions, il est défini comme le résultat du calcul d'un plus petit point fixe. Mais illustrons tout cela par un exemple.

Exemple

Reprenons notre exemple de l'atelier du chapitre 1. Le comportement du travailleur est modélisé de la manière suivante en CCS :

Travailleur <= `!in.`(*Utilisemarteau* + *Utilisemaillet*)`.`*Terminer*
Utilisemarteau <= `!prendrem.!relacherm`
Utilisemaillet <= `!prendreml.!relacherml`
Terminer <= `!out.`*Travailleur*

Ensuite, le sémaphore qui délimite une section critique pour l'allocation des outils :

Sem <= ?p.?v.*Sem*

Enfin, l'atelier qui synchronise les processus du système :

```
Atelier = ( Travailleur ||
            Travailleur ||
            Sem[?prendrem/?p,?relacherm/?v] ||
            Sem[?prendreml/?p,?relacherml/?v]
          )
          \{prendrem,relacherm,prendreml,relacherml}
```

Ce système est composé de deux systèmes de transitions : *Travailleur* et *Sem*. On les schématise ainsi :

Travailleur = $<S_T,T_T,L_T,\alpha_T,\beta_T,\lambda_T>$ où

S_T = { 0,1,2,3,4 }

$T_T = \left\{ \begin{array}{l} 0 \xrightarrow{!in} 1,1 \xrightarrow{!prendrem} 2,1 \xrightarrow{!prendreml} 3, \\ 2 \xrightarrow{!relacherm} 4,3 \xrightarrow{!relacherml} 4,4 \xrightarrow{!out} 0 \end{array} \right\}$

L_T = { !in,!out,!prendrem,!relacherm,!prendreml,!relacherml }

Sem = $<S_S,T_S,L_S,\alpha_S,\beta_S,\lambda_S>$ où

S_S = { 0,1 }

$T_S = \{0 \xrightarrow{?p} 1,1 \xrightarrow{?v} 0\}$

L_S = { ?p,?v }

On exprime l'automate *Travailleur* en Toupie comme suit :

1. On décrit l'ensemble de ses étiquettes (L_T) par le domaine

```
let travailleur_label {e,in,prendrem,relacherm
                       prendreml,relacherml,out}
```

où e symbolise l'action nulle.

2. On décrit l'ensemble de ses états (S_T) par le domaine

```
let travailleur_state {0,1,2,3,4}
```

3. Pour décrire l'ensemble des transitions, on écrit :

$$\text{Travailleur}(s,l,t) = \left\{ \bigvee_{(s_i,l_i,t_i) \in T_T} (s_i, l_i, t_i) \right\}$$

```
travailleur(S:travailleur_state,
            L:travailleur_label,
            T:travailleur_state)

           += (     %description des transitions

   ((L=e) & (S=T))
|  ((S=0) & (L=in)          & (T=1))
|  ((S=1) & (L=prendrem)    & (T=2))
|  ((S=1) & (L=prendreml)   & (T=3))
|  ((S=2) & (L=relacherm)   & (T=4))
|  ((S=3) & (L=relacherml)  & (T=4))
|  ((S=4) & (L=out)         & (T=0))
)

travailleur_initial(S:travailleur_state) += (S=0)
                    %etat de départ de l'automate.
```

Pour l'automate *Sem*, on a

```
let sem_state {0,1}
let sem_label {e,rp,rv}

sem(S:sem_state,L:sem_label,T:sem_state) += (
   ((L=e) & (S=T))
|  ((S=0) & (L=p) & (T=1))
|  ((S=1) & (L=v) & (T=0))
)

sem_initial(S:sem_state) += (S=0)
```

2.5.2 Texte MEC

En MEC, plus besoin de définir des domaines car ceux-ci sont calculés automatiquement. Donc, il suffit de décrire l'ensemble des transitions de l'automate de la manière suivante :

```
transition_system travailleur < width=0 >;
0|- in          ->1;
1|- prendrem    ->4;
4|- relacherm   ->2;
1|- prendreml   ->7;
7|- relacherml  ->2;
2|- out         ->1;
< initial = { 0 } >.
```

3. De la synchronisation au produit synchronisé

Nous décrivons cette fois-ci les étapes nécessaires pour arriver au produit synchronisé sémantiquement équivalent à la synchronisation de départ. Le but étant d'extraire tous les vecteurs de synchronisation qui vont construire le produit synchronisé. Rappelons qu'une contrainte de synchronisation est en fait l'ensemble des vecteurs de synchronisation.

3.1 Synchronisation CCS

Une synchronisation décrit le comportement d'un agent composé. Elle utilise les opérateurs : parallèle, restriction et renommage. Les symboles de parenthèses sont autorisés et on introduit une règle d'association à droite.

3.2 Arbre de synchronisation

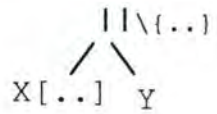
Encore une fois, c'est la structure qui se déduit le plus naturellement et le plus aisément de la description d'une synchronisation.

Définition 4.5 [Arbre de synchronisation]

Un arbre de synchronisation est un t-uple **A** = <info,R,S,f_1,f_2> où

- info est l'information contenue dans la racine
- R est l'ensemble des actions de renommage
- S est l'ensemble des actions interdites (actions de synchronisation)
- f_1,f_2 sont les arbres fils

L'opérateur `||` étant binaire, il s'agit cette fois-ci d'un arbre entièrement binaire. L'opérateur restriction fera partie de la racine de l'arbre qui représente l'expression sur laquelle il porte. Quant à l'opérateur renommage, il fera partie de la feuille de l'arbre sur laquelle il porte. Ainsi *Sync* = `(X[..] || Y)\{..}` sera représenté par l'arbre



et formellement par le t-uple *Sync* = <||,∅,{..},X,Y> où

X = <*X*,[..],∅,∅,∅>

Y = <*Y*,∅,∅,∅,∅>

Définissons de manière formelle l'équivalence sémantique entre une description de synchronisation et un arbre de synchronisation.

Nous définissons l'équivalence par induction sur la taille de la synchronisation

Définition 4.6 [Equivalence sémantique synchronisation-arbre de synchronisation]
Soit *Sync* une description de synchronisation CCS.
Soit *Sync** un arbre de synchronisation.
Alors l'arbre *Sync** est sémantiquement équivalent à la description *Sync* ssi

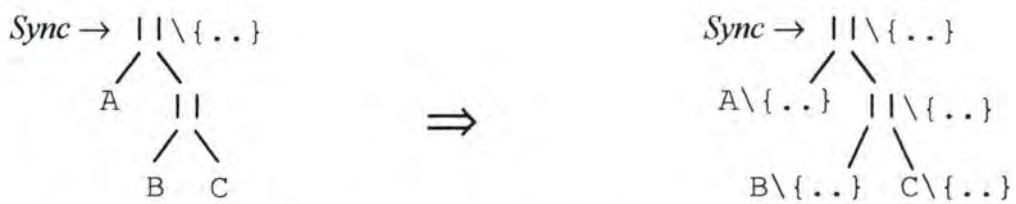
1. *Sync* → `process` et *Sync** = <process,∅,∅,∅,∅>
2. *Sync* → `process[ren]` et *Sync** = <process,ren,∅,∅,∅>
3. *Sync* → `(pr1 || pr2)\{res}` et *Sync** = <||,∅,res,pr1*,pr2*>

3.3 Arbre transformé

De l'arbre de synchronisation il serait théoriquement possible de construire le produit synchronisé et donc d'extraire les vecteurs de synchronisation. Malheureusement, en pratique, sous cette forme cela reste très fastidieux à implémenter. C'est pourquoi nous allons réaliser de petites transformations qui préservent la sémantique de l'arbre tout en l'amenant peu à peu à une forme plus facilement exploitable pour extraire les vecteurs de synchronisation.

3.3.1 Propagation des restrictions

La première transformation consiste à propager chaque restriction à l'ensemble des noeuds de l'arbre sur lequel elle porte. Pour la synchronisation *Sync* = `(A || B || C)\{..}` on schématise la transformation par :



Formellement, on définit l'équivalence sémantique entre un arbre et un arbre dans lequel on a propagé les restrictions par induction sur la structure de l'arbre.

Définition 4.7

Soit A, un arbre.

Soit S, un ensemble d'actions.

Alors l'arbre A' obtenu après propagation de S dans A est défini par les règles suivantes :

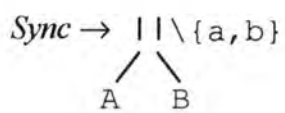
1. A = <process_name,R,∅,∅,∅> et A' = <process_name,R,S,∅,∅>

2. A = <| |,R,T,B,C> et A' = <| |,R,T $\cup_d$ S,B",C">

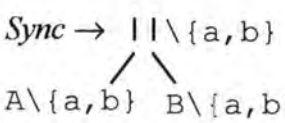
B" (resp. C") estl'arbre obtenu après propagation de T $\cup_d$ S dans B (resp. C).

Il est évident que propager les restrictions n'altère pas la sémantique de l'arbre. En effet, dans la définition 4.5 nous avions convenu d'attacher la liste des actions interdites à la racine de l'arbre de synchronisation sur lequel elles portent, signifiant par là que ces restrictions s'appliquaient à tous les processus de cet arbre. Propager ces restrictions ne fait en quelque sorte que les expliciter.

Soit la synchronisation CCS *Sync* = (A || B)\{a,b} qui interdit aux processus A et B d'effectuer les actions a et b ainsi que leur complémentaire. On schématise la synchronisation par l'arbre :



et par les t-uple <| |,∅,{a,b},A,B>, A = <A, ∅,∅,∅,∅>, B = <B, ∅,∅,∅,∅>. Après propagation, on a l'arbre schématisé par



et par les t-uple <| |,∅,{a,b},A,B>, A = <A, ∅,{a,b},∅,∅>, B = <B, ∅,{a,b},∅,∅>.

3.3.2 Transformation en liste

On transforme maintenant l'arbre obtenu en une liste reprenant les feuilles de celui-ci. Nous appellons cette liste une liste de synchronisation. Ainsi l'arbre *Sync* obtenu au point précédent devient

$$Sync \rightarrow \texttt{A\{..\}} \rightarrow \texttt{B\{..\}} \rightarrow \texttt{C\{..\}}$$

Sync est maintenant une liste des automates qui font partie du produit synchronisé.

Formellement, et par induction sur la taille de l'arbre :

Définition 4.8

Soit A, un arbre de synchronisation.
Soit L, une liste.
Alors L est sémantiquement équivalent à A

$$\Leftrightarrow$$

1. A = <process_name,R,S,∅,∅> et $L(A) = (\texttt{<process\_name,R,S,∅,∅>})$

2. A = <| |,R,S,B,C> et $L(A) = (L(B),L(C))$

Cette transformation n'altère pas la sémantique. En effet suite à la propagation des restrictions, chaque feuille de l'arbre (et donc chaque processus du système) possède l'ensemble des informations nécessaires à sa synchronisation dans le système (actions renommées et actions de restrictions). A partir de ce moment, la structure d'arbre est redondante. Changer de structure n'entraîne donc pas de perte de la sémantique.

3.4 Produit synchronisé

A partir de la liste obtenue au point précédent, on construit le produit synchronisé. Pour chaque action de chaque automate faisant partie du produit synchronisé on va construire son vecteur de synchronisation.

CCS synchronise les processus d'un système concurrent en interdisant à ceux-ci d'accomplir certaines de leurs actions individuellement. Ainsi, une action interdite ne pourra être exécutée que si, au même moment, un autre

processus du système exécute l'action complémentaire. Le mécanisme utilisé est donc l'exécution en parallèle d'actions complémentaires.

De plus, CCS possède un mécanisme de renommage des actions d'un processus, ce qui permet la mise en parallèle de plusieurs instances d'un même processus.

CCS utilise donc deux mécanismes pour synchroniser les processus d'un système concurrent : la restriction d'actions et le renommage d'actions.

Définissons maintenant l'équivalence sémantique entre une liste de synchronisation et un produit synchronisé.

Définition 4.9 [Equivalence sémantique liste-produit synchronisé]

Soit n processus P_i auxquels correspondent les automates $A_i = <S_i, T_i, L_i, \alpha_i, \beta_i, \lambda_i>$

Soit $\mathcal{L}$ une liste de synchronisation t.q. $\mathcal{L} = (AS_1,...,AS_l)$

où AS_i est l'arbre de synchronisation $<P_i, R_i, Restr_i, \varnothing, \varnothing>$.

Alors le produit synchronisé sémantiquement équivalent à $\mathcal{L}$ est le t-uple $<S, T_I, L, \alpha, \beta, \lambda>$ où

$$S = S_1 \times ... \times S_l$$
$$L = L_1 \times ... \times L_l$$
$$T_I = T_{II}$$
$$\alpha(t_1,...,t_l) = <\alpha_1(t_1),...,\alpha_l(t_l)>$$
$$\beta(t_1,...,t_l) = <\beta_1(t_1),...,\beta_l(t_l)>$$
$$\lambda(t_1,...,t_l) = <\lambda_1(t_1),...,\lambda_l(t_l)>$$

La contrainte de synchronisation, qui est l'ensemble des vecteurs de synchronisation d'un système concurrent, peut être considérée comme l'union disjointe des contraintes de synchronisation associées à chaque processus du système.

$$I = \bigcup_{i=1}^{l}{}_d\, I_i$$

Une contrainte de synchronisation associée à un processus comprend l'ensemble des vecteurs de synchronisation faisant intervenir les actions de ce processus.

$$I_i = \{V(j) \mid j \in L_i\}$$

où V(j), le vecteur de synchronisation associé à l'action j est t.q.

- si $j \notin Restr_i$ $V(j) = <act_1,...,act_{i-1},j,act_{i+1},...,act_l>$
  t. q. $act_k = e \; \forall \; k \in \{1..l\}\backslash i$

- si $j \in Restr_i$ $V(j) = <act_1,...,act_{i-1},j,act_{i+1},...,act_l>$
  t. q. $\exists!\; k \in \{1..l\}\backslash i : act_k = \overline{j}$
  et $\forall\; m \in \{1..l\}\backslash\{i,k\} : act_m = e$

3.5 Texte Toupie et texte MEC

Il s'agit maintenant de traduire le produit synchronisé. Cela revient à l'exprimer en fonction de ses vecteurs de synchronisation.

3.5.1 Texte Toupie

Un produit synchronisé s'exprime par rapport à ses vecteurs de synchronisation. En Toupie, il sera calculé à l'aide d'un plus petit point fixe. Ce calcul revient en fait à déterminer quelles actions doivent se réaliser individuellement et quelles actions doivent être synchronisées avec d'autres dans le système.

Mais notre système n'est pas encore entièrement décrit. Il faut encore calculer, toujours à l'aide d'un plus petit point fixe, l'ensemble des transitions (globales) possibles du système et ensuite, l'ensemble des états globaux accessibles à partir de l'état initial (global) et parmi les transitions autorisées. Ce qui nous donne, si nous poursuivons sur notre exemple de l'atelier :

1. Description de la contrainte de synchronisation

$$\text{Atelier}(l_1,...,l_n) = \left\{ \bigvee_{(l_{1_i},...,l_{n_i}) \in T_I} (l_{1_i},...,l_{n_i}) \right\}$$

```
atelier_synchronizator(L:travailleur_label,
                       L:travailleur_label,
                       L:sem_label,
                       L:sem_label)

                      += (

  ((L1=in)          & (L2=e)          & (L3=e) & (L4=e))
| ((L1=prendrem)    & (L2=e)          & (L3=p) & (L4=e))
```

```
| ((L1=relacherm)  & (L2=e)          & (L3=v) & (L4=e))
| ((L1=prendreml)  & (L2=e)          & (L3=e) & (L4=p))
| ((L1=relacherml) & (L2=e)          & (L3=e) & (L4=v))
| ((L1=out)        & (L2=e)          & (L3=e) & (L4=e))
| ((L1=e)          & (L2=in)         & (L3=e) & (L4=e))
| ((L1=e)          & (L2=prendrem)   & (L3=p) & (L4=e))
| ((L1=e)          & (L2=relacherm)  & (L3=v) & (L4=e))
| ((L1=e)          & (L2=prendreml)  & (L3=e) & (L4=p))
| ((L1=e)          & (L2=relacherml) & (L3=e) & (L4=v))
| ((L1=e)          & (L2=out)        & (L3=e) & (L4=e))
)
```

2. Pour calculer les transitions possibles du système, on procède récursivement. Soit X et Y une paire d'états globaux du système, alors la transition globale étiquetée par Z menant de l'état X à l'état Y est une transition autorisée dans le système si les transitions individuelles qui la composent font partie des automates du système et si l'étiquette de la transition globale Z est reprise dans les vecteurs de synchronisation.

Plus formellement :

Edge : $\mathcal{P}(S \times S) \rightarrow \mathcal{P}(S \times S)$

$\text{Edge}((a_1,b_1),...,(a_n,b_n)) = \{ \exists (l_1,...,l_n) \in L : (a_1,l_1,b_1) \in T_1 \wedge ... \wedge (a_n,l_n,b_n) \in T_n \wedge (l_1,...,l_n) \in T_I \}.$

En Toupie, on écrit la définition de plus petit point fixe

```
atelier_edge(S1:travailleur_state,T1:travailleur_state,
             S2:travailleur_state,T2:travailleur_state,
             S3:sem_state,T3:sem_state,
             S4:sem_state,T4:sem_state)

            +=

  exist L1:travailleur_label,
        L2:travailleur_label,
        L3:sem_label,
        L4:sem_label
    (
        travailleur(S1,L1,T1)
      & travailleur(S2,L2,T2)
      & sem(S3,L3,T3)
      & sem(S4,L4,T4)
      & atelier_synchronizator(L1,L2,L3,L4)
    )
```

3. Pour calculer les états globaux accessibles à partir de l'état initial, on procède récursivement. Soit X, un état global, alors X est accessible dans le système si :

- soit X est l'état initial du système

- soit il existe un état global accessible Y et une transition globale menant de l'état X à l'état Y.

Plus formellement :

Reachable : $\mathcal{P}(S) \rightarrow \mathcal{P}(S)$

$\text{Reachable}(a_1,...,a_n) = \{ \quad (\text{Initial}(a_1) \wedge ... \wedge \text{Initial}(a_n)) \vee$

$(\exists (b_1,...,b_n) \in S :$

$\text{Reachable}(b_1,...,b_n) \wedge \text{Edge}((b_1,a_1),...,(b_n,a_n)))\}$

En Toupie, on écrit la définition de plus petit point fixe

```
atelier_reachable(T1:travailleur_state,T2:travailleur_state,
                  T3:sem_state,T4:sem_state)

 += (
     (   travailleur_initial(T1=0)
       & travailleur_initial(T2=0)
       & sem_initial(T3=0)
       & sem_initial(T4=0)
     )              % etat initial du systeme

   | exist S1:travailleur_state,
          S2:travailleur_state,
          S3:sem_state,
          S4:sem_state
     (
         atelier_reachable(S1,S2,S3,S4)
                    % etat global precedent

       & atelier_edge(S1,T1,S2,T2,S3,T3,S4,T4)
                    % transition globale
     )
)
```

3.5.2 Texte MEC

En MEC l'ensemble des vecteurs de synchronisation est décrit par le système de transitions suivant :

```
synchronization_system atelier < width = 4 ;
   list = (travailleur,travailleur,sem,sem ) >;

( in          . e          . e . e );
( prendrem    . e          . p . e );
( relacherm   . e          . v . e );
( prendreml   . e          . e . p );
( relacherml  . e          . e . v );
( out         . e          . e . e );
( e           . in         . e . e );
( e           . prendrem   . p . e );
```

```
( e          . relacherm  . v . e );
( e          . prendreml  . e . p );
( e          . relacherml . e . v );
( e          . out        . e . e ).
```

Il n'y a donc plus besoin de définir le calcul des transitions permises et des états accessible du système. En effet, ceux-ci sont calculés automatiquement.
Il n'y a donc plus besoin de définir le calcul des transitions permises et des états accessibles du système. En effet, ceux-ci sont calculés automatiquement par le compilateur.

Chapitre 5

Correction* des algorithmes

Au cours du chapitre précédent, nous avons présenté les structures de représentation interne que nous avons choisies pour les processus et les synchronisations : les automates et les produits synchronisés. Nous avons aussi présenté les diverses structures intermédiaires utilisées et nous avons défini les conditions à respecter pour que ces diverses structures conservent la sémantique incluse dans le texte CCS.

Dans ce chapitre, nous nous attachons à prouver, lorsque cela est nécessaire, la correction des algorithmes qui implémentent notre compilateur. Nous prouvons donc que nos algorithmes construisent bien les structures attendues, i.e. des structures qui respectent les conditions posées au chapitre précédent et garantissent ainsi l'élaboration d'une sémantique correcte.

1. Description de processus

La description de processus CCS est transformée en un automate en plusieurs étapes. On code d'abord celle-ci sous forme d'un arbre. Ensuite on transforme l'arbre en automate puis on « réduit » cet automate. Enfin on « traduit » l'automate en code MEC et Toupie.

1.1 Du processus CCS à l'arbre

La première transformation, rappelons-le, transforme la description du processus en un arbre.

---

*Le canevas des démonstrations est celui enseigné dans le cours de *Méthodologie de la programmation* par B. Le Charlier

1.1.1 Algorithmes

Nous avons dit de la structure d'arbre qu'elle découlait naturellement et aisément de la description du processus. En effet, pour élaborer celle-ci, il suffit de parcourir la description CCS de gauche à droite, caractère par caractère, et de construire l'arbre au fur et à mesure.

Avant de faire la preuve que les algorithmes de transformation construisent bien la structure attendue, introduisons quelques notations.

1.1.1.1 Notations

Convenons de noter :

- arbre(X) : la description sémantiquement équivalente (sous forme d'arbre) de la description CCS X.
- (◊E,C) : arrivée au point E avec la condition C qui est vérifiée.
- X = `description` : X est une expression de la forme `description`.
- liste(a,...,z) : la liste composée des éléments a à z.
- arbre_bi(opérateur,opérande1,opérande2) : le constructeur d'arbre binaire <opérateur,opérande1,opérande2> (au sens de la définition 4.1).
- arbre_n(opérateur,liste_fils) : le constructeur d'arbre n-aire <opérateur,liste_fils> au sens de la définition 4.1.
- list(first,tail) : le constructeur de liste à la Lisp.
- X → Y : X pointe vers Y.

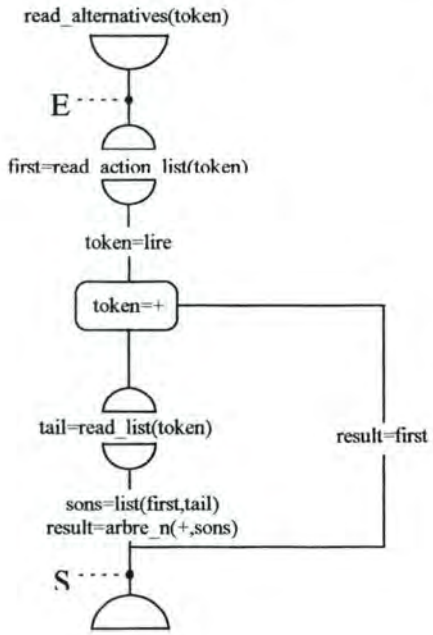
Rappellons-nous qu'une description de processus CCS peut prendre les formes suivantes (cfr chapitre 1, point 6) :

```
<process_description> ::= <alternatives>                    (1)

<alternatives> ::= <action_list> + <alternatives>           (2)
               ::= <action_list>

<action_list> ::= <action>.<action_list>                    (3a)
              ::= <process_name>.<action_list>              (3a)
              ::= (<alternatives>).<action_list>            (3a)
              ::= <action>                                  (3b)
              ::= <process_name>                            (3b)
              ::= <alternatives>                            (3b)
```

De cette syntaxe, nous déduisons trois algorithmes.

1.1.1.2 Algorithme « read_alternatives »

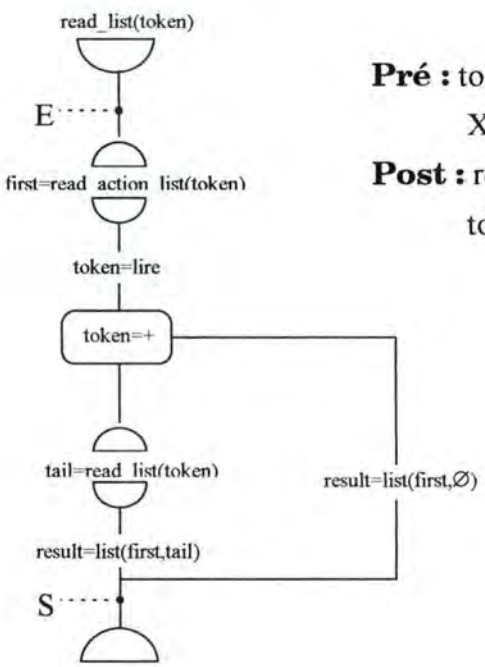
C'est l'algorithme général, qui transforme la description d'un processus CCS (sous forme (2)) en l'arbre équivalent.



Pré : token → 1er élément de X où
X est de la forme (2)
Post : result = arbre(X)
token → 1er élément après X

1.1.1.3 Algorithme « read_list »

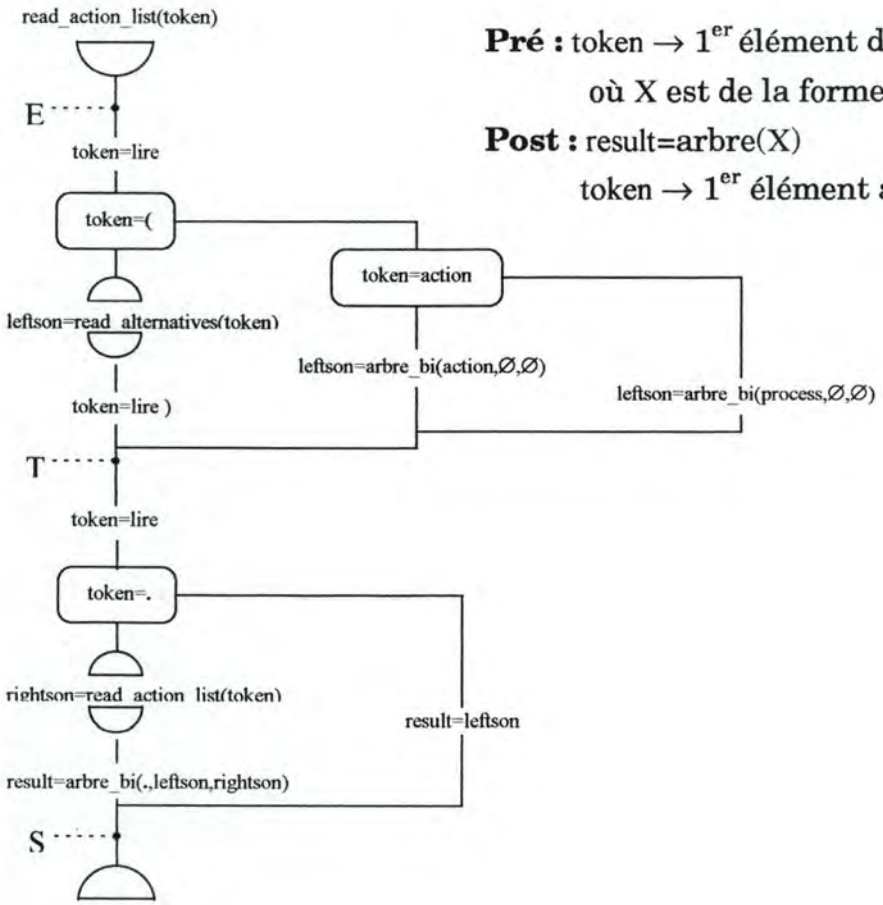
Cet algorithme transforme une série d'alternatives (forme (2)) en une liste d'arbres sémantiquement équivalents.



Pré : token → 1er élément de X où
X = <alt$_1$> + ... + <alt$_n$>
Post : result=liste(arbre(<alt$_1$>),..,arbre(<alt$_n$>))
token → 1er élément après X

1.1.1.4 Algorithme « read_action_list »

C'est l'algorithme qui transforme une suite d'actions de la forme (3) en l'arbre équivalent.



Pré : token → 1^er élément de X
où X est de la forme (3)
Post : result=arbre(X)
token → 1^er élément après X

1.1.2 Preuves

1.1.2.1 Algorithme « read_alternatives »

Prouver la correction de cet algorithme revient à prouver le théorème qui suit

Théorème 5.1

Si (◊E,token → 1^er élément de X, où X = <alternatives>)
Alors (◊S,token → 1^er élément après X,result = arbre(X))

Démonstration

Démontrons-le par induction sur la structure de X.

- cas de base : X = <action_list>

  On fait appel à read_action_list avec token → 1er élément de <action_list>. Comme les hypothèses du théorème 5.3 sont vérifiées, ⇒ au sortir de l'appel, first = arbre(<action_list>) et token → premier élément après <action_list>, c'est-à-dire que token → ∅.

  Le test échoue et on assigne first à result

  ⇒ (◊S,result = arbre(X), token → premier élément après X)

- cas inductif : X = <action_list> + <alternatives>

  Posons Y = <action_list> et Z = <alternatives>.

  On fait appel à read_action_list avec token → 1er élément de Y. Comme les hypothèses du théorème 5.3 sont vérifiées, ⇒ au sortir de l'appel, on a : first = arbre(Y) et token → premier élément après Y, soit '+' (par hypothèse).

  Le test réussit. Posons maintenant Z = <alt$_1$> + ... + <alt$_n$>. On appelle la procédure read_list avec token → Z, par le théorème 5.2, ⇒ au sortir de l'appel, on a : tail=list(arbre(<alt$_1$>),...,arbre(<alt$_n$>)) et token → ∅.

  On fait sons = list(first,tail) et result=arbre_n(+,sons).

  Or, au sens de la définition 4.2 point 4, l'arbre sémantiquement équivalent à X est l'arbre <+,<action_list>,<alt$_1$>,...,<alt$_n$>>, ce qu'établissent bien les constructeurs list et arbre_n.

  ⇒

  (◊S,result = arbre(X), token → premier élément après X)

□

1.1.2.2 Algorithme « read_list »

Il nous faut démontrer le théorème

Théorème 5.2

Si (◊E,token → 1er élément de X,X = Y_1 + ... + Y_n,Y_i = <action_list> ∀ i=1..n)

Alors

(◊S,token→premier élément après X,result = liste(arbre(Y_1),...,arbre(Y_n)))

Démonstration

Démontrons-le par induction sur la structure de X.

- cas de base : $X = Y_i$ Y_i = <action_list>

  On appelle read_action_list avec token → 1er élément de <action_list>. Comme les hypothèses du théorème 5.3 sont satisfaites, ⇒on sort de l'appel avec token → premier élément après Y_i, soit token → ∅ et first = arbre(Y_i). Ensuite le test échoue et on fait result=list(first,∅).

  ⇒(◊S,token→premier élément après Yi,result = liste(arbre(Y_i)))

- cas inductif : $X = Y_1 + ... + Y_n$ Y_i = <action_list> ∀ i=1..n

  On appelle read_action_list avec token → 1er élément de Y_1, comme les hypothèses du théorème 5.3 sont vérifiées, ⇒ on sort de l'appel avec token→premier élément après Y_1, soit token → '+' par hypothèse, et on a first = arbre(Y_1).

  Le test réussit et on appelle récursivement la procédure avec les hypothèses qui est vérifiée et $Y_2 + ... + Y_n < Y_1 + ... + Y_n$ ⇒ (par hypothèse d'induction) on sort de l'appel avec tail=liste(arbre(Y_2),...,arbre(Y_n)). On fait ensuite result=list(first,tail)

  ⇒

  (◊S,token→premier élément après Y,

  result = liste(arbre(Y_1),arbre(Y_2),...,arbre(Y_n))

□

1.1.2.3 Algorithme « read_action_list »

Pour prouver la correction de l'algorithme, démontrons le théorème qui suit.

Théorème 5.3

Si (◊E,token → 1er élément de X, où X est de la forme (3))

Alors (◊S,token → premier élément après X,result = arbre(X))

Démonstration

Par induction sur la structure de X.

- cas de base :

  1. X = <action>

     Le premier test échoue, le second réussit et on fait leftson = arbre_bi(action,∅,∅).

     ⇒

     (◊T,token → premier élément après X,leftson = <<action>,∅,∅>)

     ⇓ (définition 4.2 point 1)

(◊T,token → premier élément après X,leftson = arbre(X))

Le test échoue et on assigne leftson à result

⇒

(◊S,token → premier élément après X,result = arbre(X))

2. X = <process_name>

Le premier et le second test échouent, on fait alors leftson = arbre_bi(process,∅,∅).

⇒

(◊T,token → premier élément après X,leftson = <<process_name>,∅,∅>)

⇓ (définition 4.2 point 2)

(◊T,token → premier élément après X,leftson = arbre(X))

Le test échoue et on assigne leftson à result

⇒

(◊S,token → premier élément après X,result = arbre(X))

3. X = (Y) où Y = <alternatives>

Le premier test réussit, on appelle read_alternatives(token) avec token → 1er élément de Y :

Posons Y = W + Z où W = <action_list> et Z = <alternatives>.

Dans la procédure read_alternatives, on distingue deux cas :

- Y = W et on appelle récursivement read_action_list avec token → Y et Y < X ⇒ (par hypothèse d'induction) on sort de l'appel avec first = arbre(Y) et token → premier élément après Y.

Le test échoue et on assigne first à result.

- Y = W + Z et on appelle récursivement read_action_list avec token → W et W < X ⇒ (par hypothèse d'induction) on sort de l'appel avec first = arbre(W) et token → premier élément après W. Soit token → '+'.

Le test réussit et on appelle read_list avec token → 1er élément de Z :

Posons Z = V_i où V_i = <alternatives>. Dans la procédure read_list, on distingue deux cas :

- Z = V_1 et on appelle récursivement la procédure read_action_list avec Z < X ⇒ (par hypothèse d'induction) on sort de l'appel avec first=arbre(Z) et token → ∅. Le test échoue et on fait result = list(first,∅).

- Z = V_1 + ... + V_n et on appelle récursivement la procédure read_action_list avec token → 1er élément de V1 et V1 < X ⇒ (par hypothèse d'induction) on sort de l'appel avec first = arbre(V_1) et

token → '+'. Le test réussit et la procédure s'appelle récursivement avec $V_2 + ... + V_n < Z \Rightarrow$ on sort de l'appel avec tail=liste(arbre(V_2),...,arbre(V_n)) et token → ∅. On fait result=list(first,tail).

Dans les deux cas,

(◊S,result = liste(arbre(Z)),token → 1er élément après Z).

On sort de l'appel à read_list avec tail = liste(arbre(Z)) et on fait sons=list(first,tail) et result = arbre_n(+,sons), ce qui construit bien l'arbre sémantiquement équivalent à Y (selon la définition 4.2 point 4).

Dans les deux cas (◊S,result = arbre(Y),token → 1er élément après Y)
On sort de l'appel à read_alternatives avec leftson = arbre(Y) et token → 1er élément après Y. Le test échoue et on assigne leftson à result.

⇒ (◊S,result = arbre(X),token → 1er élément près Y)

- cas inductif X = Y.<action_list>

où Y = {<action>,<process_name>,<alternatives>}

Par la même démarche qu'au cas de base,
(◊T,leftson = arbre(Y),token → 1er élément après Y).
Soit token → '+' par hypothèse. Le test réussit et on apelle la procédure récursivement avec <action_list> < X ⇒ (par hypothèse d'induction) au sortir de l'appel on a token → premier élément après <action_list> et rightson = arbre(<action_list>). On fait result = arbre_bi(.,leftson,rightson) .
⇒
(◊S,token → premier élément après X, result=<.,leftson,rightson>)

⇓ (définition 4.2 point 3)

(◊S,token → premier élément après X, result = arbre(X))

□

1.2 De l'arbre à l'automate

Grâce à la transformation qui suit, nous arrivons à une première structure d'automate dans laquelle nous avons introduit de la redondance.

1.2.1 Algorithmes

1.2.1.1 Notations

Convenons de noter :

- create_state(val) : le créateur d'états dont la valeur est val.
- create_label(label) : le créateur d'étiquettes d'automate dont l'identificateur est label.
- create_transition(source,label,destination) : le créateur de transitions allant de l'état source dans l'état destination et étiquetée par label.
- value(state) : la fonction qui renvoie la valeur de l'état state.
- automaton ← (source,label,destination) : la fonction qui ajoute la transition (source,label,destination) dans l'automate automaton si elle ne s'y trouve pas déjà. Si automaton = $<S,T,L,\alpha,\beta,\lambda>$, on obtient $<S \cup_d \{\text{source,destination}\},T \cup_d (\text{source,label,destination}),L \cup_d \text{label},\alpha,\beta,\lambda>$.
- automaton ← state : la fonction qui ajoute l'état state dans l'automate automaton si'il ne s'y trouve pas déjà. Si automaton = $<S,T,L,\alpha,\beta,\lambda>$, on obtient $<S \cup_d \text{state},T,L,\alpha,\beta,\lambda>$.
- initial_state(X) : la fonction qui renvoie l'état initial (entree(X)) de l'automate X.
- automate(X) : l'automate sémantiquement équivalent à l'arbre X.

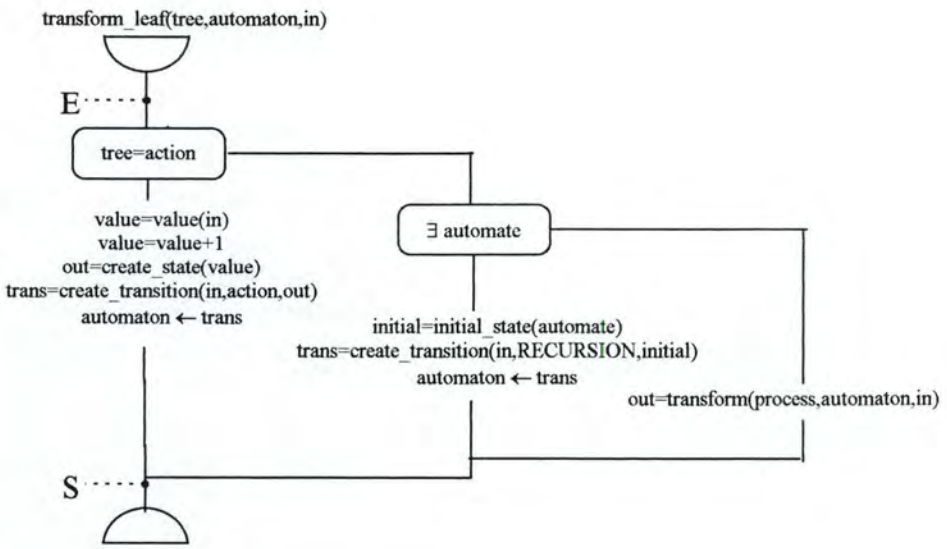
Conformément à la définition 4.3 où nous envisageons trois « formes » d'arbre, nous avons construit trois algorithme qui s'occupent chacun de la transformation d'une forme en l'automate sémantiquement équivalent.

1.2.1.2 Algorithme « transform_leaf »

Cet algorithme transforme les feuille de l'arbre en l'automate équivalent.

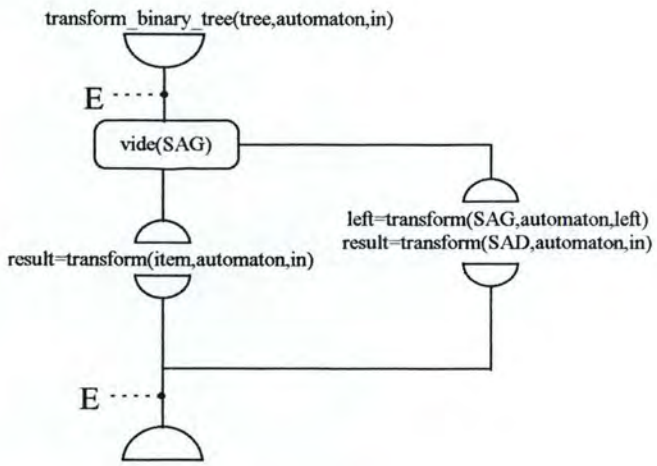
Pré : automaton=automate_0
in=entree(tree) et in $\in$ automaton
tree=feuille (au sens de la définition 4.3)

Post : automaton=$\text{automate}_0 \cup$automate(tree)
out=sortie(tree)



1.2.1.3 Algorithme « transform_binary_tree »

Cet algorithme transforme un arbre binaire en l'automate équivalent.

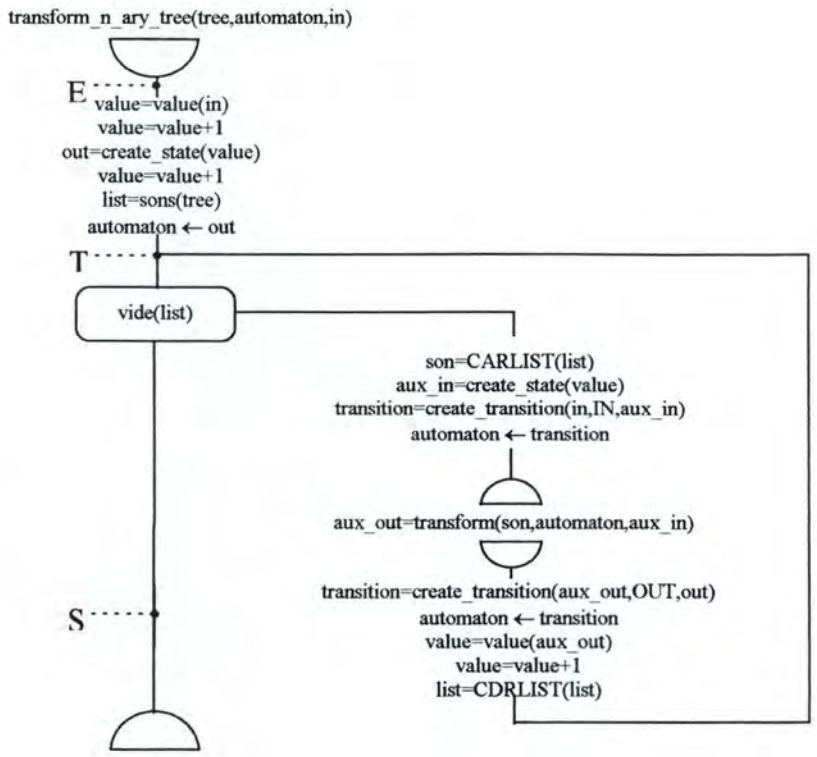


Pré : automaton=automate$_0$.
in=entree(tree).
tree=arbre binaire (au sens de la définition 4.3)

Post : automaton=automate$_0$ ∪automate(tree)
out=sortie(tree)

1.2.1.4 Algorithme « transform_n_ary_tree »

Cet algorithme transforme un arbre en l'automate équivalent.



Pré : automaton=automate$_0$.
in=entree(tree).
tree=arbre n-aire (au sens de la définition 4.3)

Post : automaton=automate$_0$ ∪automate(tree)
out=sortie(tree)

1.2.2 Preuves

1.2.2.1 Algorithme « transform_leaf »

On peut reformuler la spécification sous la forme du

Théorème 5.4

Si (◊E,tree=feuille,in=entree(tree),automaton$_0$=<S_0,T_0,L_0,α_0,β_0,λ_0>)
Alors (◊,S,out=sortie(tree),automate=automaton$_0$ ∪ automate(tree))

Démonstration

1. tree=<`action`,∅>

Le premier test réussit et on crée l'état out t.q. value(out)=value(in)+1. On fait ensuite trans=create_transition(in,`action`,out) et automaton ← trans.
⇒

($\Diamond$S,automate = < S_0,T_0,L_0,α_0,β_0,λ_0>$\cup_d$ <{in,out},trans,`action`,α,β,λ>,
out=sortie(tree))

$\Downarrow$ (par la définition 4.3)

($\Diamond$S,automate=<S_0,T_0,L_0,α_0,β_0,λ_0>$\cup_d$ automate(tree),out=sortie(tree))

2. tree=<`process`,$\varnothing$>

Le premier test échoue. Si le deuxième test réussit (on est dans le cas du B du point 1 de la définition 4.3) on cherche l'état initial de l'automate et on fait trans=create_transition(in,RECURSION,initial) et automaton $\leftarrow$ trans.

$\Rightarrow$

($\Diamond$S,automate =

<S_0,T_0,L_0,α_0,β_0,λ_0> $\cup_d$ <{in,initial},trans,RECURSION,α,β,λ>,out=sortie(tree))

$\Downarrow$ (par la définition 4.3)

($\Diamond$S,automate = <S_0,T_0,L_0,α_0,β_0,λ_0>$\cup_d$ automate(tree),out=sortie(tree))

Si le deuxième test échoue (cas C) on distingue deux cas :

- On appelle transform_binary_tree(`process`,automaton$_0$,in). Comme les hypothèses du théorème 5.5 sont vérifiées $\Rightarrow$ on sort de l'appel et

($\Diamond$S,automate = <S_0,T_0,L_0,α_0,β_0,λ_0> $\cup_d$ <S_{tree},T_{tree},L_{tree},α_{tree},β_{tree},λ_{tree}>,out=sortie(tree))

$\Downarrow$ (par la définition 4.3)

($\Diamond$S,automate=<S_0,T_0,L_0,α_0,β_0,λ_0>$\cup_d$ automate(tree),out=sortie(tree))

- On appelle transform_n_ary_tree(`process`,automaton$_0$,in), or comme les hypothèse du théorème 5.6 sont vérifiées

$\Rightarrow$

($\Diamond$S,automate=<S_0,T_0,L_0,α_0,β_0,λ_0> $\cup_d$ <S_{tree},T_{tree},L_{tree},α_{tree},β_{tree},λ_{tree}>,out=sortie(tree))

$\Downarrow$ (par la définition 4.3)

($\Diamond$S,automate=<S_0,T_0,L_0,α_0,β_0,λ_0> $\cup_d$ automate(tree),out=sortie(tree))

□

1.2.2.2 Algorithme « transform_binary_tree »

On peut reformuler la spécification sous la forme du

Théorème 5.5

Si ($\Diamond$E,tree=arbre binaire,in=entree(tree),automaton$_0$ = <S_0,T_0,L_0,α_0,β_0,λ_0>)

Alors ($\Diamond$,S,out=sortie(tree),automate=automaton$_0$ $\cup$ automate(tree))

Démonstration

Par induction sur la structure de l'arbre.

- cas de base : tree=<info,∅>

  Le test réussit et on appelle la procédure transform_leaf avec (info,automaton$_0$,in).

  Dans transform_leaf, on distingue trois cas :

  1. info = <`action`,∅>

     Le premier test réussit et on crée l'état out t.q. value(out)=value(in)+1. On fait ensuite trans=create_transition(in,`action`,out) et automaton ← trans.

     ⇒

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$>$\cup_d$ <{in,out},trans,`action`,α,β,λ>,
     out=sortie(info))

     ⇓ (par la définition 4.3)

     (◊S,automate=<$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$>$\cup_d$ automate(info),out=sortie(info))

  2. info = <`process`,∅>

     Le premier test échoue. Si le deuxième test réussit (on est dans le cas du B du point 1 de la définition 4.3) on cherche l'état initial de l'automate et on fait trans=create_transition(in,RECURSION,initial) et automaton ← trans.

     ⇒

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ <{in,initial},trans,RECURSION,α,β,λ>,
     out=sortie(info))

     ⇓ (par la définition 4.3)

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$>$\cup_d$ automate(info),out=sortie(info))

     Si le deuxième test échoue (cas C) on distingue deux cas :

     - On appelle transform_binary_tree(`process`,automaton$_0$,in)

     ⇒ (par hypothèse d'induction)

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ <$S_{info}, T_{info}, L_{info}, \alpha_{info}, \beta_{info}, \lambda_{info}$>,
     out=sortie(info))

     ⇓ (par la définition 4.3)

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$>$\cup_d$ automate(info),out=sortie(info))

     - On appelle transform_n_ary_tree(`process`,automaton$_0$,in), or comme les hypothèse du théorème 5.6 sont vérifiées

     ⇒

     (◊S,automate = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ <$S_{info}, T_{info}, L_{info}, \alpha_{info}, \beta_{info}, \lambda_{info}$>,
     out=sortie(info))

     ⇓ (par la définition 4.3)

     (◊S,automate=<$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$>$\cup_d$ automate(info),out=sortie(info))

On sort de l'appel à transform_leaf avec
automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(info) et result=out=sortie(info).
$\Rightarrow$
($\Diamond$S,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(info) et result=sortie(info)).

- cas inductif : tree=<info,SAG,SAD>
  Le test échoue et on appelle soit transform_binary_tree(SAG,automaton$_0$,in), soit transform_n_ary_tree(SAG,automaton$_0$,in).
  Si on appelle transform_binary_tree(SAG,automaton$_0$,in), comme SAG < tree $\Rightarrow$ (par hypothèse d'induction) on sort de l'appel avec left = sortie(SAG) et automaton$_1$ = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(SAG).
  Si on appelle transform_n_ary_tree(SAG,automaton$_0$,in), comme les hypothèses du théorème 5.6 sont vérifiées $\Rightarrow$ on sort de l'appel avec left=sortie(SAG) et automaton$_1$=$<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(SAG).

  Dans les deux cas on sort de l'appel avec left=sortie(SAG) et
  automaton$_1$ = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(SAG).
  On appelle ensuite transform_binary_tree(SAD,automaton,left), soit transform_n_ary_tree(SAG,automaton$_1$,left). De la même manière que ci-dessus, on sort de l'appel avec result=sortie(SAD) et
  automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(SAG) $\cup_d$ automate(SAD).
  $\Rightarrow$
  ($\Diamond$S, automaton=$<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(tree),result=sortie(tree)).

□

1.2.2.3 Algorithme « transform_n_ary_tree »

On peut reformuler la spécification sous la forme du

Théorème 5.6

Si ($\Diamond$E,tree=arbre n-aire,in=entree(tree),automaton$_0$ = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$)
Alors ($\Diamond$S,out=sortie(tree),automate=automaton$_0$ $\cup$ automate(tree))

Démonstration

Démontrons le théorème par induction sur la taille de l'arbre.

- cas de base : tree=<info,∅>

  On crée l'état out et on fait automaton ← out ⇒

  (◊T,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$ $\cup_d$ <out,∅, ∅,α,β,λ>

  Le test réussit, et

  (◊S,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$ $\cup_d$ <out,∅, ∅,α,β,λ>

  ⇓ (par la définition 4.3)

  (◊S,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$ $\cup_d$ automate(tree)).

- cas inductif : tree=<info,sons> où sons = first,sons1
  où first et sons1 sont des listes et
  first=son_1

  On crée l'état out et on fait automaton ← out ⇒

  (◊T,$automaton_1$=$<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$ $\cup_d$ <out,∅, ∅,α,β,λ>)

  Prouvons maintenant que :

  Si (◊T,sons=liste d'arbres,in=entree(tree),out=sortie(tree),
  $automaton_1$=$<S_1, T_1, L_1, \alpha_1, \beta_1, \lambda_1>$)

  Alors (◊S,in et out inchangés,automate=$automaton_1$ ∪ automaton(sons))

  Prouvons-le par induction sur la taille de la liste restant à traiter.

  Le cas de base étant trivial, passons directement au cas inductif.

  Le test échoue et on crée l'état aux_in et la transition (in,IN,aux_in). On fait automaton ← transition. On distingue alors deux cas :

  1. On appelle transform_binary_tree avec son_1, $automaton_1$ = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0>$ $\cup_d$ <{out,aux_in},transition, IN,α,β,λ> et aux_in.

     Soit son_1=<info,∅,∅> : le test réussit et on appelle la procédure transform_leaf avec (info,$automaton_1$,aux_in). Or comme les hypothèses du théorème 5.4 sont vérifiées ⇒ on sort de l'appel à transform_leaf avec automaton = $automaton_1$ $\cup_d$ automate(info) et result=sortie(info).

     ⇒

     (◊S,automaton = $automaton_1$ $\cup_d$ automate(son_1) et result=sortie(son_1)).

     Soit son_1 = <info,SAG,SAD> : le test échoue et on appelle soit transform_binary_tree(SAG,$automaton_1$,aux_in),
     soit transform_n_ary_tree(SAG,$automaton_1$,aux_in).

Si on appelle transform_binary_tree(SAG,automaton$_1$,aux_in), comme SAG < son$_1$ et que les hypothèses du théorème 5.5 sont vérifiées ⇒ (par la 1ère hypothèse d'induction) on sort de l'appel avec left=sortie(SAG) et automaton$_2$ = automaton$_1$ $\cup_d$ automate(SAG).

Si on appelle transform_n_ary_tree(SAG,automaton$_1$,aux_in), comme SAG < sons ⇒ (par la 1ère hypothèse d'induction) on sort de l'appel avec left=sortie(SAG) et automaton$_2$=automaton$_1$ $\cup_d$ automate(SAG).

Dans les deux cas on sort de l'appel avec left=sortie(SAG) et automaton$_2$=automaton$_1$ $\cup_d$ automate(SAG).

On appelle ensuite transform_binary_tree(SAD,automaton$_2$,left), soit transform_n_ary_tree(SAD,automaton$_2$,left). De la même manière que ci-dessus, on sort de l'appel avec result=sortie(SAD) et automaton = automaton$_1$ $\cup_d$ automate(SAG) $\cup_d$ automate(SAD).

⇒

(◊S, automaton = automaton$_1$ $\cup_d$ automate(son$_1$),result=sortie(son$_1$)).

2. On appelle transform_n_ary_tree avec son$_1$, automaton$_1$ = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ <{out,aux_in},transition, IN,α,β,λ> et aux_in. Comme les hypothèses sont vérifiées et que son$_1$ < tree ⇒ (par la 1ère hypothèse d'induction) on sort de l'appel avec automaton = automaton$_1$ $\cup_d$ automate(son$_1$) et aux_out=sortie(son$_1$).

Dans les deux cas, on sort de l'appel avec automaton$_2$=automaton$_1$ $\cup_d$ automate(son$_1$) et aux_out=sortie(son$_1$).

On fait transition=create_transition(aux_out,OUT,out) suivi de automaton ← transition. On passe à l'élément suivant de la liste (sons=CDRLIST(sons)), soit sons$_1$ la liste restante,

(◊T,automaton$_3$ = automaton$_2$ $\cup_d$ <{aux_out,out},(aux_out,OUT,out),OUT,α,β,λ>, aux_out=sortie(son$_1$)).

Or automaton$_2$ = automaton$_1$ $\cup_d$ automate(son$_1$) et

automaton$_1$ = <$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ <{out,aux_in},transition, IN,α,β,λ>

⇓

(◊T,automaton$_3$ =

<$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$

<{aux_in,aux_out,out},{(in,IN,aux_in),(aux_out,OUT,out)},{IN,OUT},α,β,λ> $\cup_d$ automate(son$_1$),aux_out=sortie(son$_1$)).

⇓(par la définition 4.3)

(◊T,automaton$_3$=<$S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0$> $\cup_d$ automate(first),aux_out=sortie(first)).

Comme $sons_1 < sons \Rightarrow$ (par la seconde hypothèse d'induction)

($\Diamond$S,automaton = $automaton_3 \cup_d$ automate($sons_1$),aux_out=sortie($sons_1$))

$\Downarrow$

($\Diamond$S,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(sons),aux_out=sortie(sons)).

$\Downarrow$

($\Diamond$S,automaton = $<S_0, T_0, L_0, \alpha_0, \beta_0, \lambda_0> \cup_d$ automate(tree),aux_out=sortie(sons)).

□

1.3 De l'automate à l'automate réduit

Cette transformation consiste à réduire l'automate, c'est-à-dire à enlever toute la redondance introduite lors de la transformation précédente.

1.3.1 Algorithme « reduce_automaton »

Cet algorithme fonctionne en deux étapes. Premièrement, on parcourt la liste des transitions. Chaque fois qu'on rencontre une transition étiquetée par un symbole spécial :

1. IN

   Alors la transition (source,IN,destination) devient (source,IN,destination')

   où destination' est telle que : value(destination') = value(source)
   status(destination')=MERGED.

2. OUT ou RECURSION

   Alors la transition (source,OUT (resp. RECURSION),destination) devient (source',OUT (resp. RECURSION),destination)

   où source' est telle que : value(source') = value(destination)
   status(source')=MERGED.

En marquant certains états fusionnés (MERGED) comme expliqué ci-dessus, on voit clairement qu'on construit l'ensemble $R1_Q$ (cfr définition 4.4).

La deuxième étape consiste à parcourir :

1. la liste des transitions à laquelle on enlève toutes celles étiquetées par $R2_Q$.
2. la liste des états à laquelle on enlève tous ceux qui ont fait l'objet d'une fusion (status=MERGED), i.e. qui appartiennent à $R1_Q$.
3. la liste des étiquettes à laquelle on enlève toutes celles appartenant à $R3_Q$.

1.3.2 Preuve

De l'explication du point précédent, il apparaît assez clairement que l'on construit bien l'automate réduit sémantiquement équivalent à l'automate de départ.

1.4 De l'automate réduit aux codes Toupie et MEC

L'algorithme réalisant la traduction en MEC (resp.en Toupie) ne présente aucune difficulté. Il reçoit en paramètre une structure d'automate qu 'il exploite. Il exprime les différents éléments de l'automate (états, étiquettes, transitions) en respect de la syntaxe de MEC (resp. Toupie).

2. Description de synchronisation

La description de synchronisation CCS est transformée en un produit synchronisé. Plusieurs étapes sont nécessaires pour effectuer cette transformation en conservant la sémantique de la synchronisation. On la transforme tout d'abord en un arbre de synchronisation, ensuite on transforme celui-ci pour l'amener sous forme d'une liste. Puis, de la liste on extrait les vecteurs de synchronisation qui construisent le produit synchronisé. Enfin, on peut « traduire » celui-ci en Toupie ou en MEC.

2.1 De la synchronisation à l'arbre

2.1.1 Algorithmes

Pour élaborer cette structure il suffit de parcourir la description CCS, de gauche à droite, caractère par caractère et de construire l'arbre en même temps.

2.1.1.1 Notations

Convenons de noter :

- arbre_s(opérateur,opérande1,opérande2) : le constructeur d'arbre de synchronisation <opérateur,∅,∅,opérande1,opérande2> au sens de la définition 4.5.

Une description de synchronisation peut prendre les formes suivantes :

```
<synchronized_product_description>
            ::= (<synchronized_process_list>)\
                                <synchronization_action_list>
```

(1)

```
<synchronized_process_list>
        ::= <synchronized_product_description>
                                    ||<synchronized_process_list>
        ::= <relabeled_process>||<synchronized_process_list>
        ::= <synchronized_product_description>
        ::= <relabeled_process>
```

(2)

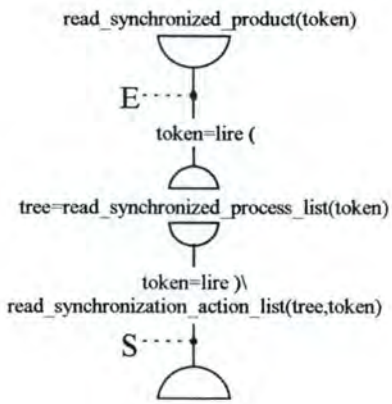
```
<relabeled_process> ::= <process_name>
                    ::= <process_name>[<relabeling_list>]
```

(3)

Nous avons déduit trois algorithmes de cette syntaxe.

2.1.1.2 Algorithme « read_synchronized_product »

Cet algorithme transforme une description de synchronisation (forme (1)) en l'arbre sémantiquement équivalent.

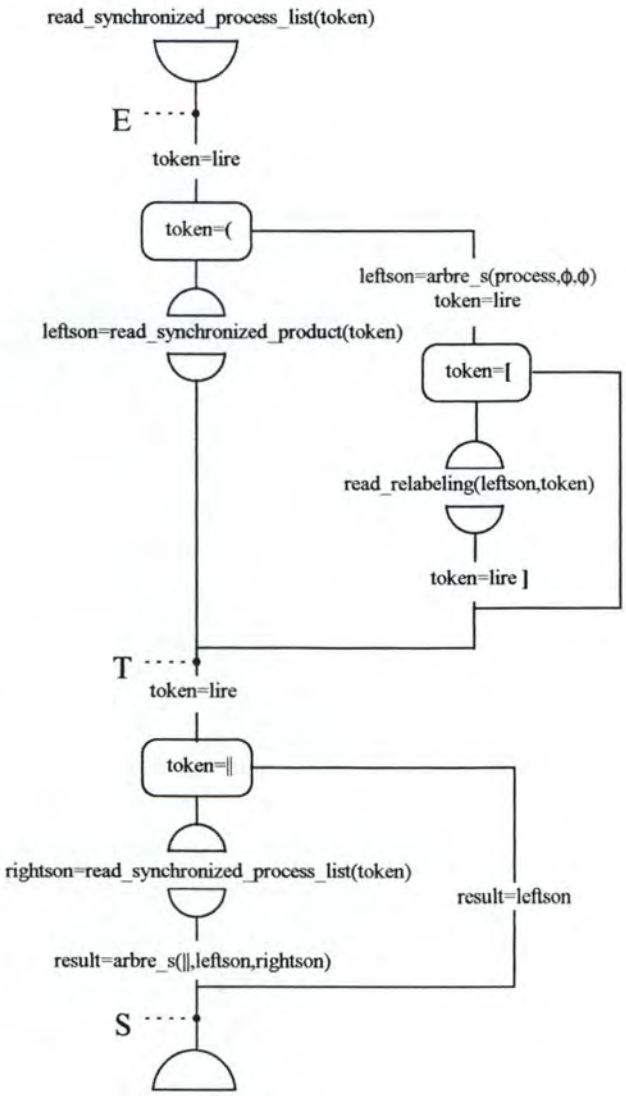


Pré : token → 1^er élément de X
où X est de la forme (1)
Post : result = arbre(X)
token → 1^er élément après X

2.1.1.3 Algorithme « read_synchronized_process_list »

Cet algorithme transforme une description de synchronisation (forme (2)) en l'arbre sémantiquement équivalent.

Pré : token → 1^er élément de X
où X est de la forme (2)
Post : result = arbre(X)
token → 1^er élément après X



2.1.1.4 Algorithme « read_relabeling »

Cet algorithme gère le renommage attaché à un processus. Sa spécification est la suivante :

Pré : token → 1^er élément de X
où X = <relabeling_list>
tree = <Y,∅,∅,∅,∅>
où Y est un processus quelconque

Post : tree = <Y,X,∅,∅,∅>
token → 1^er élément après X

2.1.1.5 Algorithme « read_synchronization_action_list »

Cet algorithme gère la restriction affectée à une synchronisation. Il est spécifié de la manière suivante :

Pré : token → 1er élément de X
où X = <synchronization_action_list>
tree = <| |,∅,∅,A,B>
où A et B sont des arbres de synchronisation quelconques

Post : tree = <| |,∅,X,A,B>
token → 1er élément après X

2.1.2 Preuves

2.1.2.1 Algorithme « read_synchronized_product »

Théorème 5.7

Si (◊E,token → 1er élément de X,X de la forme (1))
Alors (◊S,token → 1er élément après X,result=arbre(X))

Prouver la correction de l'algorithme « read_synchronized_product » revient à démontrer ce théorème.

Démonstration

Posons X=(Y)\Z où Y = <synchronization_process_list>
où Z = <synchronization_action_list>

On lit '('. On fait appel à read_synchronized_process_list avec token → 1er élément de Y. Comme les hypothèses du théorème 5.8 sont vérifiées ⇒ on sort de l'appel avec tree = arbre(Y) et token → 1er élément après Y, soit token → ')'.
On lit ')\' et on appelle read_synchronization_action_list avec token → 1er élément de Z et tree = arbre(Y) (<| |,∅,∅,...,...>). Comme sa précondition est vérifiée, on sort de l'appel avec tree = <| |,∅,Z,...,...> et token → 1er élément après Z.

⇓ (par la définition 4.6)

(◊S,tree = arbre(X),token → 1er élément après X)

□

2.1.2.2 Algorithme « read_synchronized_process_list »

Prouvons le théorème suivant :

Théorème 5.8

Si (◊E,token → 1^er élément de X,X de la forme (2))
Alors (◊S,token → 1^er élément après X,result=arbre(X))

Démonstration

Par induction sur la structure de X.

- cas de base :
  1. X = <relabeled_process>
     Le test échoue et on fait leftson=arbre_s(<process_name>,∅,∅).
     Si le processus fait l'objet d'un renommage, on appelle read_relabeling avec token → 1^er élément du renommage et leftson = <<process_name>,∅,∅,∅,∅> qui vérifient la précondition. Sinon, on ne fait rien. Dès lors, dans les deux cas
     (◊T,leftson = <<process_name>,<relabeling_list>,∅,∅,∅>,token → 1^er élément après X).
     Le test échoue et on assigne leftson à result
     ⇒
     (◊S,result = <<process_name>,<relabeling_list>,∅,∅,∅>,token → 1^er élément après X).
     ⇓ (par la définition 4.6)
     (◊S,result = arbre(X),token → 1^er élément après X).

  2. X = <synchronized_product_description>
     Posons X=(Y)\Z où Y = <synchronization_process_list>
     où Z = <synchronization_action_list>

     Le test réussit et on appelle read_synchronized_product avec token → 1^er élément de X.
     On lit '('. On fait appel à read_synchronized_process_list avec token → 1^er élément de Y. Comme Y < X ⇒ (par hypothèse d'induction) on sort de l'appel avec tree = arbre(Y) et token → 1^er élément après Y, soit token → ')'.
     On lit ')\' et on appelle read_synchronization_action_list avec token → 1^er élément de Z et tree = arbre(Y) (<| |,∅,∅,...,...>). Comme sa précondition

est vérifiée, on sort de l'appel avec tree = <| |,∅,Z,...,...> et token → 1^er élément après Z.

⇓ (par la définition 4.6)

(◊S,tree = arbre(X),token → 1^er élément après X)

On sort de l'appel et (◊T,leftson = arbre(X),token → 1^er élément après X).
Le test échoue et on assigne leftson à result
⇒
(◊S,result = arbre(X),token → 1^er élément après X).

- cas inductif : X = Y | | <synchronized_process_list>
  où Y = {<relabeled_process>,
  <synchronized_product_description>}

Quel que soit Y, par les points 1 et 2 du cas de base,
(◊T,leftson = arbre(Y),token → 1^er élément après Y).
Par hypothèse, token → '| |'. Donc le test réussit et la procédure s'appelle récursivement avec token → <synchronized_process_list>. Or comme <synchronized_process_list> < X ⇒ (par hypothèse d'induction) on sort de l'appel avec rightson = arbre(<synchronized_process_list>) et token → ∅. On fait result=arbre_s(||,leftson,rightson)

⇓ (par la définition 4.6)

(◊S,result = arbre(X),token → 1^er élément après X).

□

2.1.2.3 Algorithme « read_relabeling »

Cet algorithme ne présentant aucune difficulté, nous n'en faisons pas la preuve. Il lit simplement la liste des actions de renommage et ajoute les substitutions, i.e. les paires d'actions (action originale, action renommée) dans une liste qui est attachée à la feuille de l'arbre comportant le processus sur lequel elles portent.

2.1.2.4 Algorithme « read_synchronization_action_list »

Pas de preuve non plus. L'algorithme lit simplement la liste des actions interdites et ajoute celles-ci dans une liste qui est attachée à la racine de l'arbre sur lequel elles portent.

2.2 De l'arbre à la liste de synchronisation

On effectue ce passage en deux étapes : on « propage » les restrictions puis on crée une liste avec les feuilles de l'arbre.

2.2.1 Algorithme « propagate_restrictions »

Cet algorithme « propage » les restrictions appliquées à un arbre parmi l'ensemble des noeuds de cet arbre. Il fonctionne comme suit :

Il reçoit en paramètre la liste des actions de restriction à propager, soit P ainsi que l'arbre dans lequel il faut les propager.

- si arbre = $\varnothing$
  rien à faire
- si arbre = feuille (soit <feuille,R,$\varnothing$,$\varnothing$,$\varnothing$>)
  on ajoute l'ensemble des actions à la feuille $\Rightarrow$ <feuille,R,P,$\varnothing$,$\varnothing$>

- si arbre = <| |,$\varnothing$,S,A,B>
  on ajoute l'ensemble des actions qui ne se trouve pas encore dans l'arbre.
  $\Rightarrow$ <| |,$\varnothing$,S $\cup_d$ P,A,B>
  on appelle récursivement la procédure sur A avec Q = S $\cup_d$ P.
  on appelle récursivement la procédure sur B avec Q = S $\cup_d$ P.

Il est évident que cet algorithme respecte la définition 4.7.

2.2.2 Algorithme « flatten_tree »

Cet algorithme chaîne les feuilles d'un arbre de synchronisation en une liste de synchronisation. Il suffit de parcourir l'arbre dans l'ordre infixé et d'ajouter en fin de liste chaque feuille rencontrée.

L'algorithme fonctionne comme suit :

- si arbre = feuille
  on ajoute la feuille à la liste

- si arbre = <| |,R,S,SAG,SAD>
  on appelle récursivement la procédure sur SAG.
  on appelle récursivement la procédure sur SAD.

Cet algorithme est fidèle à la définition 4.8.

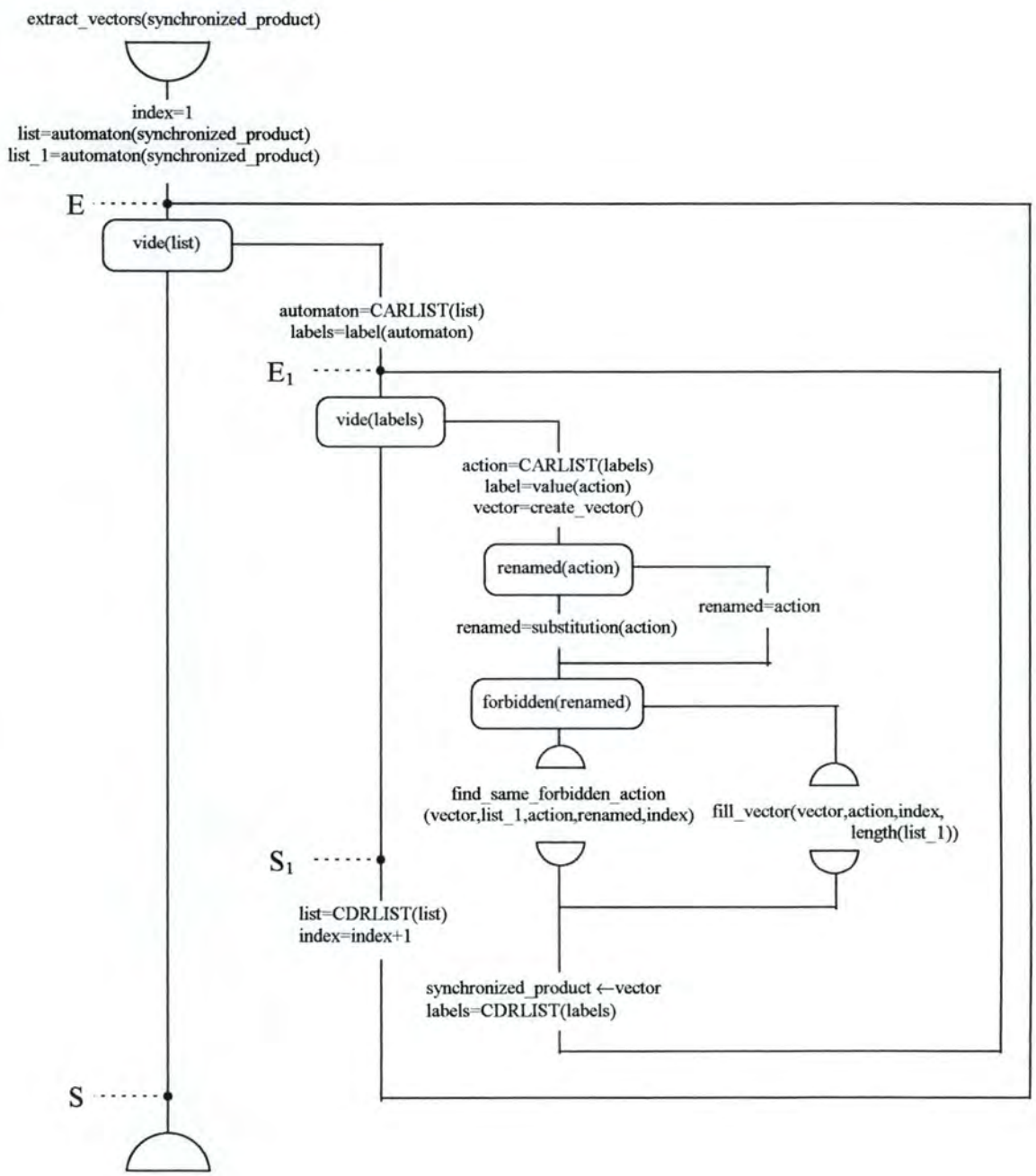
2.3 De la liste de synchronisation au produit synchronisé

Dans cette dernière étape, on extrait les vecteurs de synchronisation qui constituent le produit synchronisé.

2.3.1 Algorithmes

2.3.1.1 Algorithme « extract_vectors »

Cet algorithme extrait l'ensemble des vecteurs de synchronisation « contenus » dans une liste de synchronisation.



Pré : list initialisé,synchronized_product = <S,∅,L,α,β,λ>)

Post :synchronized_product= <S,T_I,L,α,β,λ>
où I est la contrainte de synchronisation associée à list.

2.3.1.2 Algorithme « fill_vector »

Cet algorithme génère un vecteur de synchronisation caractérisant une action qui ne doit être synchronisée avec aucune autre.

Pré : vector,action,length initialisés.
index initialisé t.q. $1 \leq index \leq length$
Post : vector = $<a_1,...,a_{index-1},action,a_{index+1},...,a_{length}>$
où $a_i = e \; \forall \; i \in \{1..length\} \backslash index$

2.3.1.3 Algorithme « find_same_forbidden_action »

Etant donnée une action d'un processus d'un système concurrent, cet algorithme cherche avec quelle action de quel processus elle doit se synchroniser et génère le vecteur de synchronisation correspondant.

Pré : vector,action,renamed initialisés.
index initialisé t.q. $1 \leq index \leq length$
Post : $\exists \; i \in \{1..length\} \backslash index$:
vector = $<a_1,...,a_{i-1},\overline{action},a_{i+1},...,a_{index-1},action,a_{index+1},...,a_{length}>$
où $a_i = e \; \forall \; i \in \{1..length\} \backslash \{index,i\}$

2.3.1.4 Notations

Convenons de noter :

- label(automaton) : la fonction qui renvoie sous forme d'une liste les étiquettes d'un automate.
- synchronized_product ← vector : la fonction qui ajoute le vecteur vector au produit synchronise synchronized_product s'il ne s'y trouve pas déjà.
- create_vector : le créateur de vecteur de synchronisation.
- value(label) : la fonction qui renvoie la valeur de l'étiquette label.
- substitution(action) : la fonction qui renvoie l'action qui renomme action.

- contrainte(list) : la contrainte de synchronisation sémantiquement équivalente à la liste list.

2.3.2 Preuves

2.3.2.1 Algorithme « extract_vectors »

Théorème 5.9

Si ($\Diamond$E,list=liste de synchronisation,synchronized_product = <S,$\varnothing$,L,α,β,λ>)
Alors ($\Diamond$S,synchronized_product= <S,T_I,L,α,β,λ>,I = contrainte(list))

Il convient tout d'abord de démontrer le lemme suivant.

Lemme

Si ($\Diamond E_1$,tree = <A_i,R_i,$Restr_i$,$\varnothing$,$\varnothing$>,synchronized_product = <S,T_I,L,α,β,λ>,
labels=liste d'actions)
Alors ($\Diamond S_1$,synchronized_product = <S,$T_{I \cup_d I_{A_i}}$,L,α,β,λ>)

Démonstration

Démontrons-le par induction sur la taille de la liste restant à traiter.

- cas de base : labels = $\varnothing$
  On ne fait rien et $\Diamond S_1$ avec la thèse.

- cas inductif : labels = (act_1,...,act_n)
  Le test échoue et on crée le vecteur vide.
  Si l'action act_1 fait l'objet d'un renommage, on fait renamed=substitution(act_1), sinon on assigne act_1 à renamed.
  On distingue alors deux cas :
  1. act_1 est interdite, on appelle find_same_forbidden_action avec sa précondition qui est vérifiée $\Rightarrow$ on sort de l'appel avec
  vector = <...,act_1,...,$\overline{act_1}$,...>
  2. act_1 n'est pas interdite, on appelle fill_vector avec sa précondition qui est vérifiée $\Rightarrow$ on sort de l'appel avec vector = <...,act_1,...>

  Dans les deux cas, on sort de l'appel avec vector = vector(act_1)
  On fait synchronized_product $\leftarrow$ vector et on passe à l'élément suivant de la liste.

Soit labels1 = (act_2,...,act_n) la liste restante.

($\Diamond E_1$,synchronized_product = <S,$T_{I \cup_d vector(act_1)}$,L,α,β,λ>,labels1)

Or labels1 < labels ⇒ (par hypothèse d'induction)

($\Diamond S_1$,synchronized_product = <S,$T_{I \cup_d vector(act_1) \cup_d ... \cup_d vector(act_n)}$,L,α,β,λ>)

⇓ (par la définition 4.9)

($\Diamond S_1$,synchronized_product = <S,$T_{I \cup_d I_{A_i}}$,L,α,β,λ>)

□

Démonstration du théorème 5.9

Par induction sur la longueur de la liste restant à traiter.

- cas de base : list = ∅

  Le test réussit et ◊S avec la thèse.

- cas inductif : list = (AS_1,...,AS_n) où AS_i = <A_i,R_i,$Restr_i$,∅,∅>

  On initialise index à 1. Le test échoue et on fait labels=label(A_i).

  ◊E1 avec tree = <A_1,R_1,$Restr_1$,∅,∅>et labels. Comme les hypothèses du lemme sont vérifiées ⇒

  (◊S1 ,synchronized_product = <S,$T_{I_{A_1}}$,L,α,β,λ>,labels1)

  On incrémente index de 1 et on passe à l'élément suivant de la liste.

  Soit list1 = (AS_2,...,AS_n) la liste restante.

  (◊E,synchronized_product= <S,$T_{I_{A_1}}$,L,α,β,λ>,list1). Or list1 < list ⇒ (par hypothèse d'induction)

  (◊S,synchronized_product = <S,$T_{I_{A_1} \cup_d ... \cup_d I_{A_n}}$,L,α,β,λ>)

  ⇓ (par la définition 4.9)

  (◊S,synchronized_product = <S,T_I,L,α,β,λ,>,I = contrainte(list))

□

2.4 Du produit synchronisé aux codes Toupie et MEC

L'algorithme réalisant la traduction en MEC (resp.en Toupie) ne présente aucune difficulté. Il reçoit en paramètre une structure de produit synchronisé qu'il exprime en respect de la syntaxe de MEC (resp. Toupie).

Chapitre 6

Exemples

Ce chapitre a un double but. Premièrement présenter les problèmes engendrés par la concurrence ainsi que leur modélisation. Deuxièmement, présenter plus en détail deux exemples. C'est pourquoi, après avoir défini les problèmes qui peuvent survenir dans un système concurrent, nous donnons deux exemples illustrant ces problèmes. Nous les présentons, donnons leur modélisation CCS. Nous montrons ensuite explicitement les différentes structures utilisées pour extraire la sémantique et nous donnons enfin le code Toupie et le code MEC.

1. Les problèmes engendrés par la concurrence

Puisqu'un système concurrent consiste en l'exécution en parallèle de plusieurs processus interagissant et partageant des ressources communes, il est naturel qu'il survienne des problèmes. On en distingue trois principaux.

1.1 Le deadlock (interblocage)

Il s'agit d'une situation où un processus est définitivement bloqué en attente d'une ressource possédée par un autre processus lui-même définitivement bloqué en attente d'une ressource possédée par le premier. En termes d'états et de transitions, un deadlock est un état dans lequel aucune transition n'est possible ou seulement des transitions menant à un état de deadlock.

1.2 La starvation (famine)

La famine survient lorsque plusieurs processus se partagent une ressource critique. Il se peut qu'un processus ne parvienne pas à acquérir cette

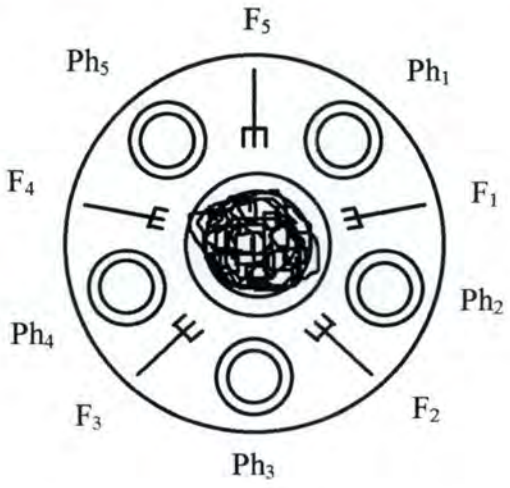
ressource (parce que les autres processus sont plus rapides, par exemple). Ce processus est alors dans un état de famine c'est-à-dire un état d'attente perpétuelle.

1.3 Le livelock

Dans un système concurrent, le livelock survient lorsque des processus se partagent une ressource critique. Des processus sont dans un état de livelock lorsque, d'une part ils essaient d'entrer dans leur section critique et n'y parviennent pas, d'autre part s'ils ne restent pas inactifs (ils essaient réellement d'entrer dans leur section critique). Le livelock est une attente active.

2. Le dîner des philosophes

Cinq philosophes ont pris place autour d'une table circulaire. Ils passent leur temps à penser mais, bien sûr, il leur faut aussi manger occasionnellement s'il veulent continuer à vivre et à penser. Au milieu de la table se trouve un vaste plat de spaghetti. Pour manger les spaghettis il leur faut deux fourchettes. On a disposé le couvert de telle sorte qu'il n'y ait que cinq fourchettes, chaque convive ne pouvant se servir que des deux fourchettes placées respectivement à la droite et à la gauche de son assiette.



Cet exemple sert à illustrer le fait qu'un système où le philosophe acquiert une seule fourchette à la fois peut conduire à des situations d'interblocage (deadlock). C'est le cas, par exemple, si tous les philosophes ont saisi leur fourchette gauche.

Mais cet exemple sert surtout à mettre en évidence le phénomène d'attente indéfinie ou famine (starvation). Un philosophe pourrait attendre indéfiniment si quelques convives se liguaient contre lui, ou plus simplement, si

ses voisins sont de gros mangeurs et sont plus prompts que lui à saisir les fourchettes.

Nous présentons un algorithme (tiré de [Ram90]) qui résout ces deux problèmes.

```
sémaphore s init 4;
tableau [0..4] sf sémaphore init (1,1,1,1,1)

procédure philosophe(i);
début répéter
        début
            penser;
            p(s);
            p(sf[i]);
            p(sf[(i+1) mod 5]);
            manger;
            v(sf[i]);
            v(sf[(i+1) mod 5]);
            v(s);
        fin
fin
```

Grâce au sémaphore sf, on introduit une exclusion mutuelle sur l'allocation et la désallocation des fourchettes empêchant ainsi l'interblocage. Quant au sémaphore s, il permet de ne laisser entrer que quatre philosophes dans la section critique qu'il détermine empêchant ainsi tout problème de famine.

2.1 Le modélisation en CCS

Le comportement d'une fourchette est le suivant : à l'origine elle est sur la table, un philosophe s'en saisit (`sf`), l'utilise puis la repose (`lf`). Ce que nous modélisons ainsi :

Fork <= `?sf.?lf.`*Fork*

Le philosophe passe son temps à penser (`think`). Lorsqu'il veut manger, il prend sa fourchette gauche (`sfg`) puis sa fourchette droite (`sfd`), il mange (`eat`) puis les repose (`rfg` et `rfd`). Ensuite il se remet à penser. Nous modélisons ce comportement par :

Philosophe <= `!think.!in.!sfg.!sfd.!eat.!lfg.!lfd.!out.`*Philosophe*

Finalement, le sémaphore s qui ne laisse rentrer que quatre personnes dans la section critique fonctionne comme suit : lorsqu'il n'y a personne dans la section critique, le sémaphore est dans son état initial. Si une personne rentre (`?in`), le sémaphore passe dans un état tel qu'il ne peut laisser entrer que trois personnes (état initial de *Sem3*) ou laisser sortir celle qui est entrée (`?out`). S'il entre une autre personne, il se retrouve dans un état ou seulement deux personnes peuvent encore entrer (état initial de *Sem2*) ; au contraire, si la personne sort, il se retrouve dans un état où il peut de nouveau laisser entrer une personne de plus (ici l'état initial). Si quatre personnes sont entrées, le sémaphore doit d'abord en laisser sortir une avant de pouvoir en admettre une nouvelle.

On le modélise par le processus *Sem4* :

Sem1 <= `?in.?out.`*Sem1* + `?out`
Sem2 <= `?in.`*Sem1.Sem2* + `?out`
Sem3 <= `?in.`*Sem2.Sem3* + `?out`
Sem4 <= `?in.`*Sem3.Sem4*

On synchronise le système par la déclaration

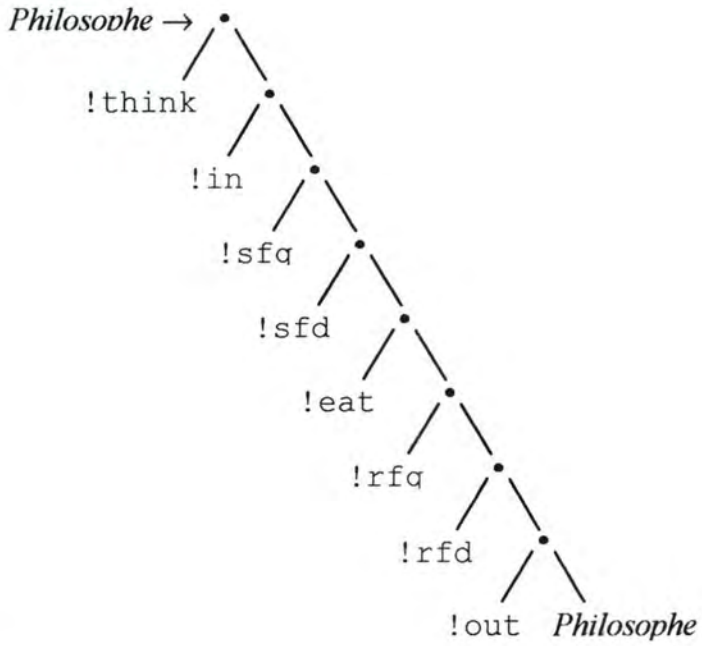
```
Dining = ( Philosophe[!sf1/!sfg,!sf5/!sfd,!lf1/!lfg,!lf5/!lfd]   ||
           Philosophe[!sf2/!sfg,!sf1/!sfd,!lf2/!lfg,!lf1/!lfd]   ||
           Philosophe[!sf3/!sfg,!sf2/!sfd,!lf3/!lfg,!lf2/!lfd]   ||
           Philosophe[!sf4/!sfg,!sf3/!sfd,!lf4/!lfg,!lf3/!lfd]   ||
           Philosophe[!sf5/!sfg,!sf4/!sfd,!lf5/!lfg,!lf4/!lfd]   ||

           Fork[?sf1/?sf,?lf1/?lf]                               ||
           Fork[?sf2/?sf,?lf2/?lf]                               ||
           Fork[?sf3/?sf,?lf3/?lf]                               ||
           Fork[?sf4/?sf,?lf4/?lf]                               ||
           Fork[?sf5/?sf,?lf5/?lf]                               ||

           Sem4
         )
             \{in,out,sf1,sf2,sf3,sf4,sf5,lf1,lf2,lf3,lf4,lf5}
```

Les actions entre accolades sont les actions de synchronisation. Un philosophe ne pourra entrer dans la section critique en exécutant `!in` que si le sémaphore *Sem4* peut accomplir `?in` en même temps. De même, un philosophe ne pourra saisir sa fourchette que si elle n'est pas déjà en possession de son voisin, i.e. *Philosophe*

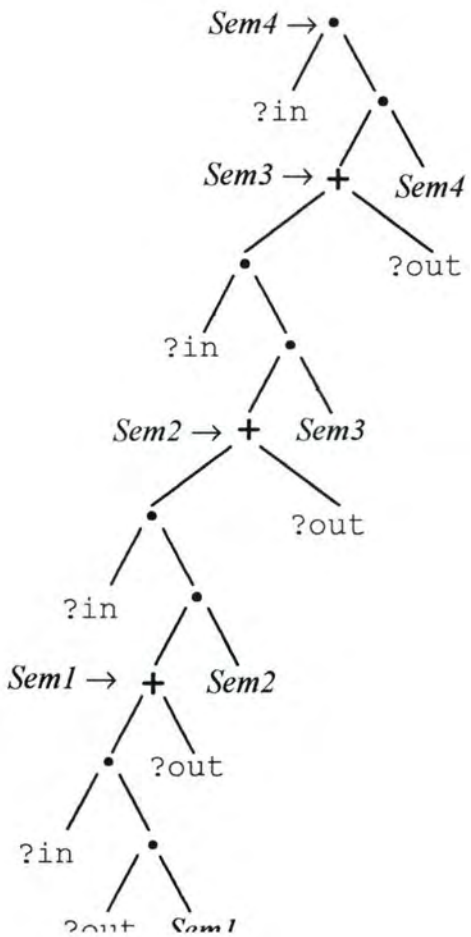
pourra accomplir `!sfg` que si le sémaphore *Fork* symbolisant sa fourchette gauche peut accomplir en même temps `?sfg`.

2.2 Les arbres

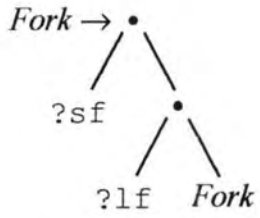
Le philosophe :



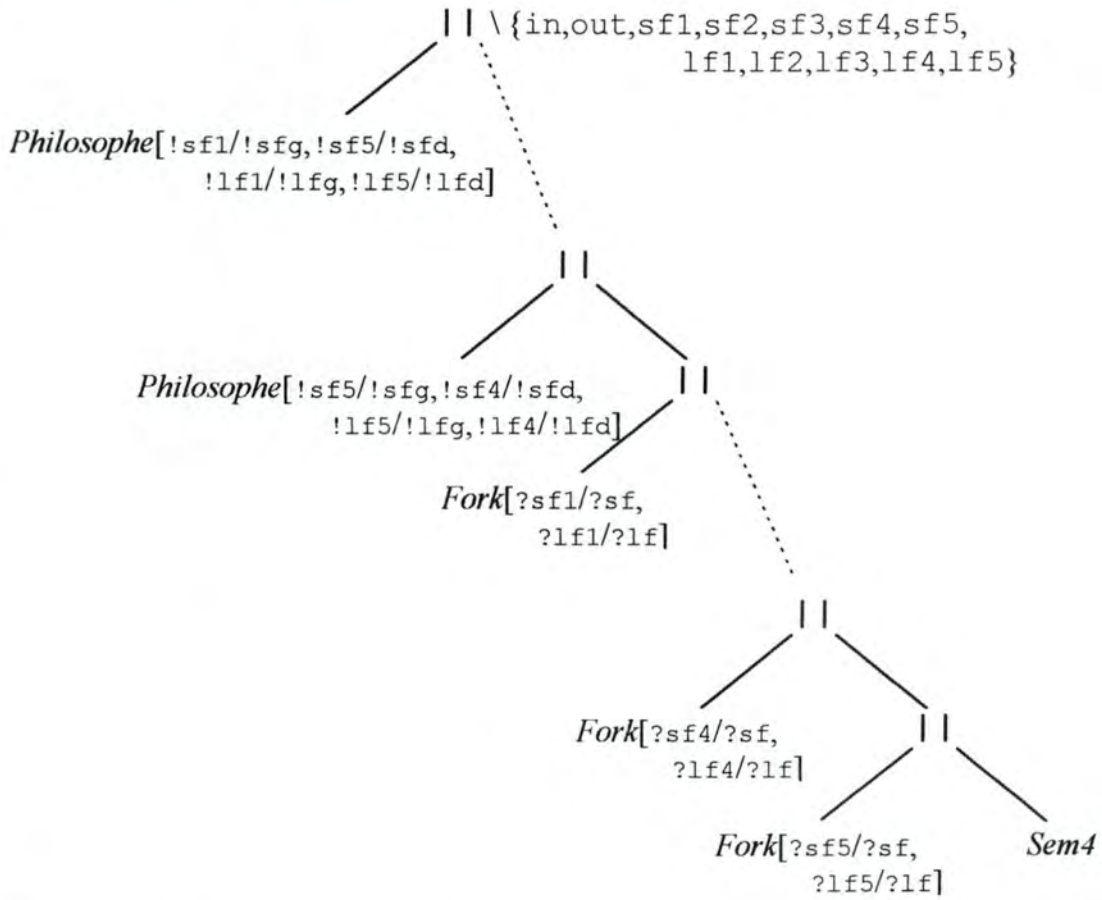
Le sémaphore s :



La fourchette :

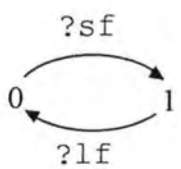


La synchronisation du problème :

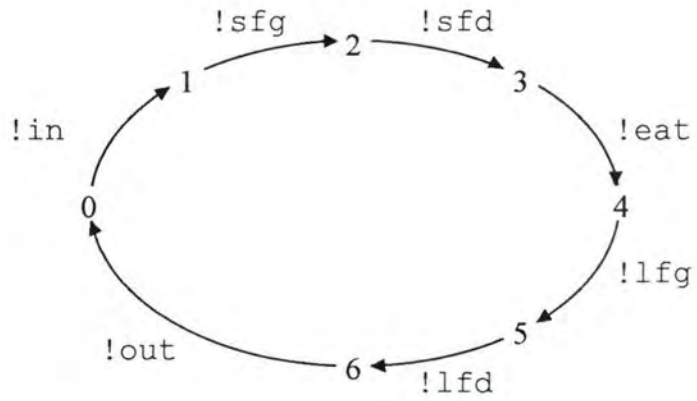


2.3 Les automates réduits

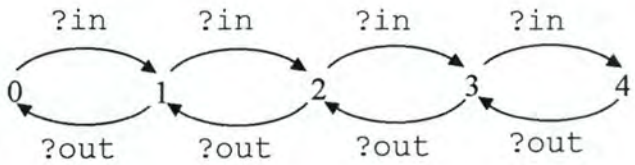
La fourchette :



Le philosophe :



Le sémaphore s :



2.4 Le code Toupie

Les déclarations qui suivent expriment l'automate philosophe.

```
let philosophe_state {0,1,2,3,4,5,6,7}

let philosophe_label
                {e,sthink,sin,ssfg,ssfd,seat,slfg,slfd,sout}

philosophe(S@1:philosophe_state,L@2:philosophe_label,
                                        T@3:philosophe_state) += (
     ((L=e) & (S=T))
   | ((S=0) & (L=sthink) & (T=1))
   | ((S=1) & (L=sin)    & (T=2))
   | ((S=2) & (L=ssfg)   & (T=3))
   | ((S=3) & (L=ssfd)   & (T=4))
   | ((S=4) & (L=seat)   & (T=5))
   | ((S=5) & (L=slfg)   & (T=6))
   | ((S=6) & (L=slfd)   & (T=7))
   | ((S=7) & (L=sout)   & (T=0))
 )

philosophe_initial(S@1:philosophe_state) += (S=0)
```

On décrit ensuite la fourchette.

```
let fork_state {0,1}

let fork_label {e,rsf,rlf}
```

```
fork(S@4:fork_state,L@5:fork_label,T@6:fork_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=rsf) & (T=1))
  | ((S=1) & (L=rlf) & (T=0))
 )

fork_initial(S@4:fork_state) += (S=0)
```

Enfin, on décrit le sémaphore

```
let sem4_state {0,1,2,5}
let sem4_label {e,rin,rout}

sem4(S@7:sem4_state,L@8:sem4_label,T@9:sem4_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=rin)  & (T=1))
  | ((S=1) & (L=rin)  & (T=2))
  | ((S=2) & (L=rin)  & (T=3))
  | ((S=3) & (L=rin)  & (T=4))
  | ((S=4) & (L=rout) & (T=3))
  | ((S=3) & (L=rout) & (T=2))
  | ((S=2) & (L=rout) & (T=1))
  | ((S=1) & (L=rout) & (T=0))
 )

sem4_initial(S@7:sem4_state) += (S=0)
```

On synchronise le système en décrivant l'ensemble des vecteurs d'actions autorisés.

```
dining_synchronizator(L1@2:philosophe_label,
                      L2@5:philosophe_label,
                      L3@8:philosophe_label,
                      L4@11:philosophe_label,
                      L5@14:philosophe_label,
                      L6@17:fork_label,
                      L7@20:fork_label,
                      L8@23:fork_label,
                      L9@26:fork_label,
                      L10@29:fork_label,
                      L11@32:sem4_label) += (
   ((L1=sthink) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=sin) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rin))
 | ((L1=ssfg) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=rsf) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=ssfd) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=rsf) & (L11=e))
 | ((L1=seat) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=slfg) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=rlf) &
```

```
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=slfd) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=e) & (L8=e) & (L9=e) & (L10=rlf) & (L11=e))
 | ((L1=sout) & (L2=e) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rout))
 | ((L1=e) & (L2=sthink) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=sin) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rin))
 | ((L1=e) & (L2=ssfg) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
   (L7=rsf) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=ssfd) & (L3=e) & (L4=e) & (L5=e) & (L6=rsf) &
   (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=seat) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=slfg) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=rlf) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=slfd) & (L3=e) & (L4=e) & (L5=e) & (L6=rlf) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=sout) & (L3=e) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rout))
 | ((L1=e) & (L2=e) & (L3=sthink) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=sin) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rin))
 | ((L1=e) & (L2=e) & (L3=ssfg) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=rsf) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=ssfd) & (L4=e) & (L5=e) & (L6=e) &
    (L7=rsf) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=seat) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=slfg) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=rlf) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=slfd) & (L4=e) & (L5=e) & (L6=e) &
    (L7=rlf) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=sout) & (L4=e) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rout))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=sthink) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=sin) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rin))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=ssfg) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=rsf) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=ssfd) & (L5=e) & (L6=e) &
    (L7=e) & (L8=rsf) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=seat) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=slfg) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=rlf) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=slfd) & (L5=e) & (L6=e) &
    (L7=e) & (L8=rlf) & (L9=e) & (L10=e) & (L11=e))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=sout) & (L5=e) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rout))
 | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=sthink) & (L6=e) &
    (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
```

```
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=sin) & (L6=e) &
     (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rin))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=ssfg) & (L6=e) &
     (L7=e) & (L8=e) & (L9=e) & (L10=rsf) & (L11=e))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=ssfd) & (L6=e) &
     (L7=e) & (L8=e) & (L9=rsf) & (L10=e) & (L11=e))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=seat) & (L6=e) &
     (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=e))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=slfg) & (L6=e) &
     (L7=e) & (L8=e) & (L9=e) & (L10=rlf) & (L11=e))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=slfd) & (L6=e) &
     (L7=e) & (L8=e) & (L9=rlf) & (L10=e) & (L11=e))
  | ((L1=e) & (L2=e) & (L3=e) & (L4=e) & (L5=sout) & (L6=e) &
     (L7=e) & (L8=e) & (L9=e) & (L10=e) & (L11=rout)))
```

On remarque clairement ici l'intérêt d'écrire les choses en CCS puis de faire générer le code Toupie automatiquement par le compilateur.

Le prédicat suivant calcule l'ensemble des transitions globales possibles du système.

```
dining_edge(S1@1:philosophe_state,T1@3:philosophe_state,
            S2@4:philosophe_state,T2@6:philosophe_state,
            S3@7:philosophe_state,T3@9:philosophe_state,
            S4@10:philosophe_state,T4@12:philosophe_state,
            S5@13:philosophe_state,T5@15:philosophe_state,
            S6@16:fork_state,T6@18:fork_state,
            S7@19:fork_state,T7@21:fork_state,
            S8@22:fork_state,T8@24:fork_state,
            S9@25:fork_state,T9@27:fork_state,
            S10@28:fork_state,T10@30:fork_state,
            S11@31:sem4_state,T11@33:sem4_state) +=

  exist  L1@2:philosophe_label,
         L2@5:philosophe_label,
         L3@8:philosophe_label,
         L4@11:philosophe_label,
         L5@14:philosophe_label,
         L6@17:fork_label,
         L7@20:fork_label,
         L8@23:fork_label,
         L9@26:fork_label,
         L10@29:fork_label,
         L11@32:sem4_label
    (
        philosophe(S1,L1,T1)
      & philosophe(S2,L2,T2)
      & philosophe(S3,L3,T3)
      & philosophe(S4,L4,T4)
      & philosophe(S5,L5,T5)
      & fork(S6,L6,T6)
      & fork(S7,L7,T7)
      & fork(S8,L8,T8)
```

```
      & fork(S9,L9,T9)
      & fork(S10,L10,T10)
      & sem4(S11,L11,T11)
      & dining_synchronizator(L1,L2,L3,L4,L5,L6,
                              L7,L8,L9,L10,L11)
    )
```

Le prédicat qui suit calcule l'ensemble des états accessibles dans le système.

reachable (Z) = {Initial(Z) ∨ ∃ Y : (reachable(Y) ∧ edge(Y,Z))}

```
dining_reachable(T1@3:philosophe_state,
                 T2@6:philosophe_state,
                 T3@9:philosophe_state,
                 T4@12:philosophe_state,
                 T5@15:philosophe_state,
                 T6@18:fork_state,
                 T7@21:fork_state,
                 T8@24:fork_state,
                 T9@27:fork_state,
                 T10@30:fork_state,
                 T11@33:sem4_state) += (
    (philosophe_initial(T1=0) & philosophe_initial(T2=0) &
     philosophe_initial(T3=0) & philosophe_initial(T4=0) &
     philosophe_initial(T5=0) & fork_initial(T6=0) &
     fork_initial(T7=0) & fork_initial(T8=0) &
     fork_initial(T9=0) & fork_initial(T10=0) &
     sem4_initial(T11=0))

  | exist  S1@1:philosophe_state,
          S2@4:philosophe_state,
          S3@7:philosophe_state,
          S4@10:philosophe_state,
          S5@13:philosophe_state,
          S6@16:fork_state,
          S7@19:fork_state,
          S8@22:fork_state,
          S9@25:fork_state,
          S10@28:fork_state,
          S11@31:sem4_state
     (
        dining_reachable(S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
      & dining_edge(S1,T1,S2,T2,S3,T3,S4,T4,S5,T5,
                    S6,T6,S7,T7,S8,T8,S9,T9,S10,T10,S11,T11)
     )
  )

dining_reachable() +=
  exist  T1@3:philosophe_state,
         T2@6:philosophe_state,
         T3@9:philosophe_state,
         T4@12:philosophe_state,
         T5@15:philosophe_state,
```

```
          T6@18:fork_state,
          T7@21:fork_state,
          T8@24:fork_state,
          T9@27:fork_state,
          T10@30:fork_state,
          T11@33:sem4_state

    dining_reachable(T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11)
```

Le prédicat suivant calcule si un état est un état de deadlock, c'est-à-dire

deadlock(X) = {reachable(X) $\wedge$ ($\forall$ Z : (reachable(Z) $\wedge$ edge(X,Z)) $\Rightarrow$ deadlock(Z))}

```
dining_deadlock(S1@1:philosophe_state,
                S2@4:philosophe_state,
                S3@7:philosophe_state,
                S4@10:philosophe_state,
                S5@13:philosophe_state,
                S6@16:fork_state,
                S7@19:fork_state,
                S8@22:fork_state,
                S9@25:fork_state,
                S10@28:fork_state,
                S11@31:sem4_state) += (

    dining_reachable(S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
  & forall  T1@3:philosophe_state,
            T2@6:philosophe_state,
            T3@9:philosophe_state,
            T4@12:philosophe_state,
            T5@15:philosophe_state,
            T6@18:fork_state,
            T7@21:fork_state,
            T8@24:fork_state,
            T9@27:fork_state,
            T10@30:fork_state,
            T11@33:sem4_state


      (
          (
              dining_reachable(T1,T2,T3,T4,T5,T6,
                                              T7,T8,T9,T10,T11)
            & dining_edge(S1,T1,S2,T2,S3,T3,S4,T4,S5,T5,S6,T6,
                          S7,T7,S8,T8,S9,T9,S10,T10,S11,T11)
          )
        => dining_deadlock(T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11)
      )
  )
```

Le prédicat qui suit calcule s'il existe des états de deadlock.

```
dining_deadlock() +=
  exist  S1@1:philosophe_state,
         S2@4:philosophe_state,
         S3@7:philosophe_state,
         S4@10:philosophe_state,
         S5@13:philosophe_state,
         S6@16:fork_state,
         S7@19:fork_state,
         S8@22:fork_state,
         S9@25:fork_state,
         S10@28:fork_state,
         S11@31:sem4_state

    dining_deadlock(S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
```

Les prédicats qui suivent calculent les états de starvation dans le système. A cet effet, on recalcule l'ensemble des états accessibles en interdisant tous les états où un processus se trouve dans sa section critique. Il n'y a pas de problèmes de famine si tous les états du nouveau produit synchronisé obtenu sont des états de deadlock. En effet, si tous les états ne sont pas deadlocks, cela signifie que certains processus continuent à évoluer alors que les autres sont bloqués devant leur section critique, ce qui est bien la définition que nous avons donnée de l'interblocage.

```
dining_reachable_without_cs
                (T1@3:philosophe_state,
                 T2@6:philosophe_state,
                 T3@9:philosophe_state,
                 T4@12:philosophe_state,
                 T5@15:philosophe_state,
                 T6@18:fork_state,
                 T7@21:fork_state,
                 T8@24:fork_state,
                 T9@27:fork_state,
                 T10@30:fork_state,
                 T11@33:sem4_state) += (
    (philosophe_initial(T1=0) & philosophe_initial(T2=0) &
     philosophe_initial(T3=0) & philosophe_initial(T4=0) &
     philosophe_initial(T5=0) & fork_initial(T6=0) &
     fork_initial(T7=0) & fork_initial(T8=0) &
     fork_initial(T9=0) & fork_initial(T10=0) &
     sem4_initial(T11=0))

  | exist  S1@1:philosophe_state,
           S2@4:philosophe_state,
           S3@7:philosophe_state,
           S4@10:philosophe_state,
```

```
            S5@13:philosophe_state,
            S6@16:fork_state,
            S7@19:fork_state,
            S8@22:fork_state,
            S9@25:fork_state,
            S10@28:fork_state,
            S11@31:sem4_state
      (

dining_reachable_without_cs(S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
       & dining_edge(S1,T1,S2,T2,S3,T3,S4,T4,S5,T5,
                     S6,T6,S7,T7,S8,T8,S9,T9,S10,T10,S11,T11)
       & (T1#4) & (T2#4) & (T3#4) & (T4#4) & (T5#4)
      )
  )

dining_deadlock_without_cs
               (S1@1:philosophe_state,
                S2@4:philosophe_state,
                S3@7:philosophe_state,
                S4@10:philosophe_state,
                S5@13:philosophe_state,
                S6@16:fork_state,
                S7@19:fork_state,
                S8@22:fork_state,
                S9@25:fork_state,
                S10@28:fork_state,
                S11@31:sem4_state) += (


dining_reachable_without_cs(S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
  & forall  T1@3:philosophe_state,
            T2@6:philosophe_state,
            T3@9:philosophe_state,
            T4@12:philosophe_state,
            T5@15:philosophe_state,
            T6@18:fork_state,
            T7@21:fork_state,
            T8@24:fork_state,
            T9@27:fork_state,
            T10@30:fork_state,
            T11@33:sem4_state

      (
          (
              dining_reachable_without_cs(T1,T2,T3,T4,T5,T6,
                                          T7,T8,T9,T10,T11)
            & dining_edge(S1,T1,S2,T2,S3,T3,S4,T4,S5,T5,S6,T6,
                          S7,T7,S8,T8,S9,T9,S10,T10,S11,T11)
            & (T1#4) & (T2#4) &(T3#4) &(T4#4) &(T5#4)
          )
        => dining_deadlock_without_cs(T1,T2,T3,T4,T5,T6,
                                      T7,T8,T9,T10,T11)
      )
  )
```

```
starvation(S1@1:philosophe_state,
           S2@4:philosophe_state,
           S3@7:philosophe_state,
           S4@10:philosophe_state,
           S5@13:philosophe_state,
           S6@16:fork_state,
           S7@19:fork_state,
           S8@22:fork_state,
           S9@25:fork_state,
           S10@28:fork_state,
           S11@31:sem4_state) +=

 ~dining_deadlock_without_cs(S1,S2,S3,S4,S5,6,
                                          S7,S8,S9,S10,S11)
```

2.5 Le code MEC

```
transition_system Philosophe < width=0 >;
0|- sthink ->1,
1|- sin    ->2;
2|- ssfg   ->3;
3|- ssfd   ->4;
4|- seat   ->5;
5|- slfg   ->6;
6|- slfd   ->7;
7|- sout   ->0;
< initial = { 0 } >.

transition_system Fork < width=0 >;
0|- rsf ->1;
1|- rlf ->0;
< initial = { 0 } >.

transition_system Sem4 < width=0 >;
0|- rin  ->1;
1|- rin  ->2;
2|- rin  ->3;
3|- rin  ->4;
4|- rout ->3;
3|- rout ->2;
2|- rout ->1;
1|- rout ->0;
< initial = { 0 } >.
```

```
synchronization_system Dining < width = 11 ;
   list = (Philosophe,Philosophe,Philosophe,
           Philosophe,Philosophe,Fork,Fork,Fork,Fork,Fork,Sem4
) >;

( sthink . e . e . e . e . e . e . e . e . e );
( sin . e . e . e . e . e . e . e . e . rin );
( ssfg . e . e . e . e . rsf . e . e . e . e );
( ssfd . e . e . e . e . e . e . e . e . rsf . e );
( seat . e . e . e . e . e . e . e . e . e . e );
( slfg . e . e . e . e . e . rlf . e . e . e . e );
( slfd . e . e . e . e . e . e . e . e . e . rlf . e );
( sout . e . e . e . e . e . e . e . e . e . rout );
( e . sthink . e . e . e . e . e . e . e . e . e );
( e . sin . e . e . e . e . e . e . e . e . rin );
( e . ssfg . e . e . e . e . rsf . e . e . e . e );
( e . ssfd . e . e . e . e . rsf . e . e . e . e );
( e . seat . e . e . e . e . e . e . e . e . e );
( e . slfg . e . e . e . e . e . rlf . e . e . e . e );
( e . slfd . e . e . e . e . rlf . e . e . e . e );
( e . sout . e . e . e . e . e . e . e . e . rout );
( e . e . sthink . e . e . e . e . e . e . e . e );
( e . e . sin . e . e . e . e . e . e . e . rin );
( e . e . ssfg . e . e . e . e . rsf . e . e . e );
( e . e . ssfd . e . e . e . e . rsf . e . e . e );
( e . e . seat . e . e . e . e . e . e . e . e );
( e . e . slfg . e . e . e . e . e . rlf . e . e );
( e . e . slfd . e . e . e . e . rlf . e . e . e );
( e . e . sout . e . e . e . e . e . e . e . rout );
( e . e . e . sthink . e . e . e . e . e . e . e );
( e . e . e . sin . e . e . e . e . e . e . rin );
( e . e . e . ssfg . e . e . e . e . rsf . e . e );
( e . e . e . ssfd . e . e . e . e . rsf . e . e );
( e . e . e . seat . e . e . e . e . e . e . e );
( e . e . e . slfg . e . e . e . e . e . rlf . e );
( e . e . e . slfd . e . e . e . e . rlf . e . e );
( e . e . e . sout . e . e . e . e . e . e . rout );
( e . e . e . e . sthink . e . e . e . e . e . e );
( e . e . e . e . sin . e . e . e . e . e . rin );
( e . e . e . e . ssfg . e . e . e . e . rsf . e );
( e . e . e . e . ssfd . e . e . e . e . rsf . e );
( e . e . e . e . seat . e . e . e . e . e . e );
( e . e . e . e . slfg . e . e . e . e . e . rlf );
( e . e . e . e . slfd . e . e . e . e . rlf . e );
( e . e . e . e . sout . e . e . e . e . e . rout ).
```

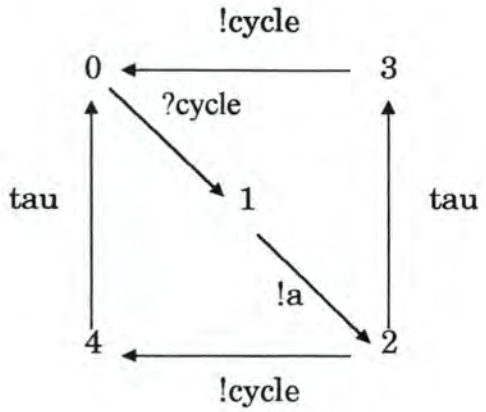
3. Le scheduler de Milner

Dans une machine monoprocesseur, le rôle d'un scheduler est de permettre l'exécution en parallèle de plusieurs processus. Pour ce faire, celui-ci donne la main, i.e. donne la permission de s'exécuter à chaque processus à tour

de rôle. Le problème se compose d'un « starter » qui est un processus qui initialise le système, et de n « cycler » qui s'exécutent en parallèle. Chaque processus « cycler » attend la permission (?cycle) pour démarrer, il effectue une action quelconque (!a) et passe le jeton (!cycle) avant ou après une action interne (tau).

En fait les processus ne s'exécutent pas véritablement en parallèle vu que la machine est monoprocesseur. Il s'agit plutôt de permettre à chaque processus de s'exécuter « un petit peu » à tour de rôle.

Le schéma d'un processus « cycler » est le suivant (c'est en même temps le schéma de son automate) :



3.1 La modélisation en CCS

Le processus « cycler » est décrit par

Cycler <= `?cycle.!a.(!cycle.tau.`*Cycler* `+ tau.!cycle.`*Cycler*)

Le processus qui initialise le système :

Start <= `!cycle.NIL`

On synchronise le système, pour n = 3 par la déclaration de synchronisation :

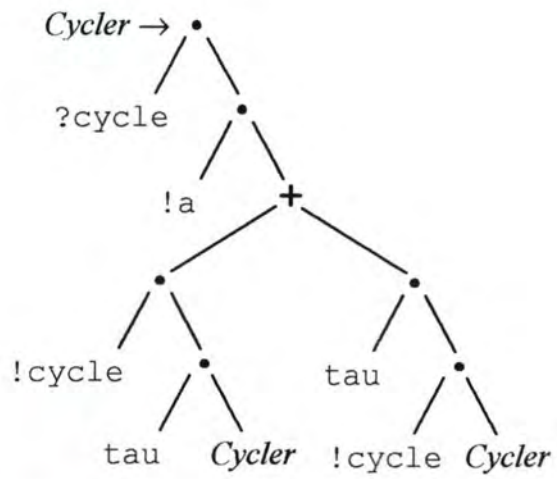
Scheduler3 =

```
(
 (
  ( Start[!cycle1/!cycle]                                    ||
    Cycler[?cycle1/?cycle,!cycle2/!cycle,!a1/!a]
  )\cycle1                                                   ||
```

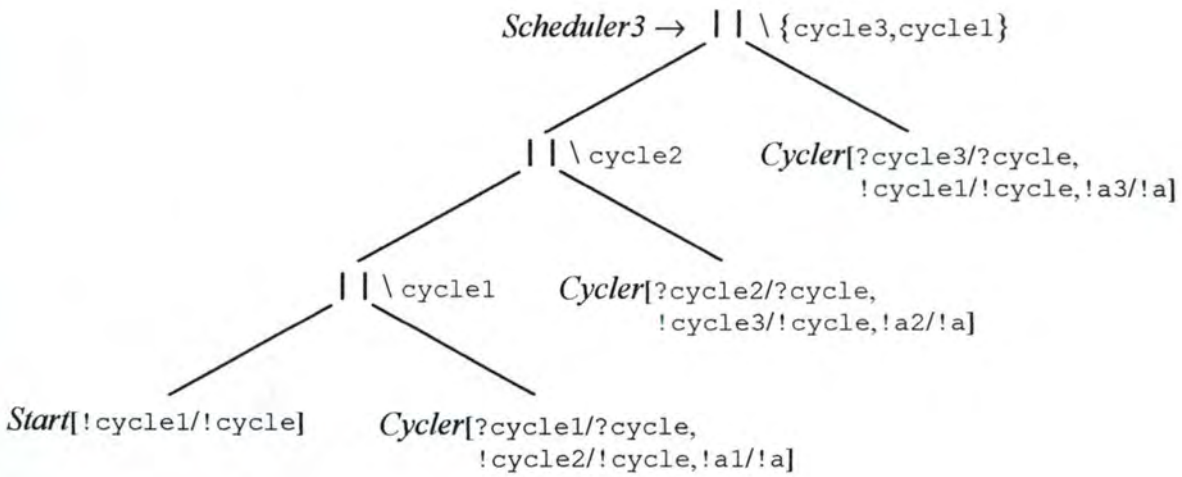
```
  Cycler[?cycle2/?cycle,!cycle3/!cycle,!a2/!a]
 )\cycle2                                                    ||

Cycler[?cycle3/?cycle,!cycle1/!cycle,!a3/!a]
)\{cycle3,cycle1}
```

3.2 Les arbres

Pour le processus cycler, on a :



Enfin, l'arbre qui synchronise les scheduler pour n=3 :



L'arbre du starter se résume à une feuille.

3.3 Le code Toupie

Le processus starter

```
let start_state {0,1}
let start_label {e,scycle}

start(S@1:start_state,L@2:start_label,T@3:start_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=scycle) & (T=1))
 )

start_initial(S@1:start_state) += (S=0)
```

Le processus cycler

```
let cycler_state {0,1,2,5,8}
let cycler_label {e,rcycle,sa,scycle,tau}

cycler(S@4:cycler_state,L@5:cycler_label,T@6:cycler_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=rcycle) & (T=1))
  | ((S=1) & (L=sa)     & (T=2))
  | ((S=2) & (L=scycle) & (T=5))
  | ((S=5) & (L=tau)    & (T=0))
  | ((S=2) & (L=tau)    & (T=8))
  | ((S=8) & (L=scycle) & (T=0))
 )

cycler_initial(S@4:cycler_state) += (S=0)
```

Le produit synchronisé

```
scheduler3_synchronizator(L1@2:start_label,
                          L2@5:cycler_label,
                          L3@8:cycler_label,
                          L4@11:cycler_label) += (

    ((L1=scycle) & (L2=rcycle) & (L3=e)      & (L4=e))
  | ((L1=e)      & (L2=sa)     & (L3=e)      & (L4=e))
  | ((L1=e)      & (L2=scycle) & (L3=rcycle) & (L4=e))
  | ((L1=e)      & (L2=tau)    & (L3=e)      & (L4=e))
  | ((L1=e)      & (L2=e)      & (L3=sa)     & (L4=e))
  | ((L1=e)      & (L2=e)      & (L3=scycle) & (L4=rcycle))
  | ((L1=e)      & (L2=e)      & (L3=tau)    & (L4=e))
  | ((L1=e)      & (L2=e)      & (L3=e)      & (L4=sa))
  | ((L1=e)      & (L2=rcycle) & (L3=e)      & (L4=scycle))
  | ((L1=e)      & (L2=e)      & (L3=e)      & (L4=tau)))
```

Calcul des transitions possibles dans le système

```
scheduler3_edge(S1@1:start_state,T1@3:start_state,
                S2@4:cycler_state,T2@6:cycler_state,
                S3@7:cycler_state,T3@9:cycler_state,
                S4@10:cycler_state,T4@12:cycler_state) +=
  exist L1@2:start_label,
        L2@5:cycler_label,
```

```
        L3@8:cycler_label,
        L4@11:cycler_label
    (
        start(S1,L1,T1)
      & cycler(S2,L2,T2)
      & cycler(S3,L3,T3)
      & cycler(S4,L4,T4)
      & scheduler3_synchronizator(L1,L2,L3,L4)
    )
```

Calcul des états accessibles dans le système

```
scheduler3_reachable(T1@3:start_state,
                     T2@6:cycler_state,
                     T3@9:cycler_state,
                     T4@12:cycler_state) += (

    (start_initial(T1=0) & cycler_initial(T2=0) &
     cycler_initial(T3=0) & cycler_initial(T4=0))

  | exist S1@1:start_state,
         S2@4:cycler_state,
         S3@7:cycler_state,
         S4@10:cycler_state
     (
        scheduler3_reachable(S1,S2,S3,S4)
      & scheduler3_edge(S1,T1,S2,T2,S3,T3,S4,T4)
     )
  )

scheduler3_reachable() +=
  exist T1@3:start_state,
        T2@6:cycler_state,
        T3@9:cycler_state,
        T4@12:cycler_state

        scheduler3_reachable(T1,T2,T3,T4)
```

Calcul des états de deadlock

```
scheduler3_deadlock(S1@1:start_state,
                    S2@4:cycler_state,
                    S3@7:cycler_state,
                    S4@10:cycler_state) += (

    scheduler3_reachable(S1,S2,S3,S4)
  & forall T1@3:start_state,
           T2@6:cycler_state,
           T3@9:cycler_state,
           T4@12:cycler_state
      (
          (
              scheduler3_reachable(S1,S2,S3,S4)
            & scheduler3_edge(S1,T1,S2,T2,S3,T3,S4,T4)
```

```
            )
          => scheduler3_deadlock(T1,T2,T3,T4)
        )
    )

scheduler3_deadlock() +=
  exist S1@1:start_state,
        S2@4:cycler_state,
        S3@7:cycler_state,
        S4@10:cycler_state

    scheduler3_deadlock(S1,S2,S3,S4)
```

Calcul des états de livelock

```
scheduler3_co_reachable(S1@1:start_state,
                        S2@4:cycler_state,
                        S3@7:cycler_state,
                        S4@10:cycler_state) += (

    (start_initial(S1=0) & cycler_initial(S2=0) &
     cycler_initial(S3=0) & cycler_initial(S4=0))

  | exist T1@3:start_state,
         T2@6:cycler_state,
         T3@9:cycler_state,
         T4@12:cycler_state
      (
          scheduler3_co_reachable(T1,T2,T3,T4)
        & scheduler3_edge(S1,T1,S2,T2,S3,T3,S4,T4)
      )
  )


scheduler3_livelock(S1@1:start_state,
                    S2@4:cycler_state,
                    S3@7:cycler_state,
                    S4@10:cycler_state) -= (

    scheduler3_reachable(S1,S2,S3,S4)
 & ~scheduler3_co_reachable(S1,S2,S3,S4)
)
```

3.4 Le code MEC

Le starter

```
transition_system Start < width=0 >;
0|- scycle ->1;
< initial = { 0 } >.
```

```
transition_system Cycler < width=0 >;
0|- rcycle ->1;
1|- sa     ->2;
2|- scycle ->5;
5|- tau    ->0;
2|- tau    ->8;
8|- scycle ->0;
< initial = { 0 } >.


synchronization_system Scheduler3 < width = 4 ;
   list = (Start,Cycler,Cycler,Cycler ) >;

( scycle . rcycle . e      . e );
( e      . sa     . e      . e );
( e      . scycle . rcycle . e );
( e      . tau    . e      . e );
( e      . e      . sa     . e );
( e      . e      . scycle . rcycle );
( e      . e      . tau    . e );
( e      . e      . e      . sa );
( e      . rcycle . e      . scycle );
( e      . e      . e      . tau ).
```

Bibliographie

[ABC94] A. Arnold, D. Begay and P. Crubillé. Construction and analysis of transition systems with MEC. April 1994

[Arn89] A. Arnold. MEC : a System for Constructing and Analysing Transitions Systems. In *Automatic Verifications Methods for Finite State Systems*, volume 407 of *LNCS*. Springer-Verlag, 1989.

[Bou93] A. Bouali. Etudes et mise en oeuvre d'outils de vérification basée sur la bisimultation. PhD thesis, Université Paris VII, 03 1993.

[Bry86] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE transactions on computer*, 35:677-691, 8 1986.

[CCMR93] M-M. Corsini, B. Le Charlier, K. Musumbu, and A. Rauzy. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Symbolic Finite Domains, Research report, University of Namur, Belgium, 1993.

[CR93a] M-M. Corsini and A. Rauzy. First Experiments with Toupie. Technical Report 577-93, LaBRI - Université Bordeaux I, 1993.

[CR93b] M-M. Corsini and A. Rauzy. Toupie User's Manual. Technical Report 586-93, LaBRI - Université Bordeaux I, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[Ram90] J. Ramaekers. *Systèmes d'exploitation*. Cours de 2ème licence, 1990

Table des matières